TAMPEREEN TEKNILLINEN YLIOPISTO

JUHA-PEKKA ARIMAA

TUOTANNONSUUNNITTELUJÄRJESTELMÄN KOMPONENTTI-POHJAINEN KEHITTÄMINEN

Diplomityö

Tarkastaja: professori Kai Koskimies
Tarkastaja ja aihe hyväksytty
Tieto- ja sähkötekniikan tiedekunnan
tiedekuntaneuvoston kokouksessa
8. kesäkuuta 2011

TIIVISTELMÄ

TAMPEREEN TEKNILLINEN YLIOPISTO
Tietotekniikan koulutusohjelma
ARIMAA, JUHA-PEKKA: Tuotannonsuunnittelujärjestelmän komponenttipohjainen kehittäminen
Diplomityö, 53 sivua, 3 liitesivua
Marraskuu 2011
Pääaine: Ohjelmistotuotanto
Tarkastaja: professori Kai Koskimies
Avainsanat: Tuotannonsuunnittelujärjestelmä, komponentti, komponenttipohjainen kehittäminen, ohjelmistoarkkitehtuuri, tuoteperhe

Komponenttipohjainen kehitys tarkoittaa ohjelmistojen rakentamista uudelleenkäytettäviä komponentteja hyväksikäyttäen. Läheisesti uudelleenkäyttöön liittyvä tekniikka on myös tuoterunkojen käyttö ohjelmistokehityksessä. Niin komponenttien kuin tuoterunkojenkin käytön tavoitteena on parantaa laatua, vähentää kustannuksia, pienentää tuotantoaikaa ja helpottaa ohjelmistokehitystä.

Työ on toteutettu yhteistyössä SW-Developmentin kanssa ja työn tavoitteena on laatia komponentteja ja tuoterunkoa hyödyntävä arkkitehtuurisuunnitelma yrityksen tuotannonsuunnittelujärjestelmille. Työ jakaantuu kolmeen osaan. Ensimmäisessä osassa käsitellään komponenttien ja tuoterunkojen teoreettista taustaa sekä aikaisempia havaintoja niiden käytöstä. Toisessa osassa tutkitaan olemassa olevan järjestelmän rakennetta ja toimintaa sekä käsitellään sen keskeisimpiä ongelmakohtia. Tutkittavan järjestelmän perusteella laaditaan myös vaatimukset suunniteltavalle järjestelmälle. Kolmannessa osassa tehdään varsinainen arkkitehtuurisuunnitelma ja toteutetaan sille systemaattinen arviointiprosessi.

Työ toimii osaltaan lähtökohtana SW-Developmentin tuleville järjestelmille. Lisäksi työ tuo esille komponentti- ja tuoterunkoarkkitehtuurien keskeisimpiä piirteitä, haasteita, hyötyjä ja ongelmakohtia.

ABSTRACT

TAMPERE UNIVERSITY OF TECHNOLOGY
Master's Degree Programme in Information Technology
ARIMAA, JUHA-PEKKA: Component-based development of Planning Efficiency System
Master of Science Thesis, 53 pages, 3 Appendix pages
November 2011
Major: Software Engineering
Examiner: Professor Kai Koskimies
Keywords: Planning Efficiency System, component, component-based development, software architecture, product family

Component-based software development means building software systems with the help of reusable components. Another technique for reusability is the use software product lines. Both component-based software development and software product lines aim at better quality, lesser costs, shorter production time and easier development process.

This thesis is done in cooperation with SW-Development and the goal of this thesis is to design an architecture which takes advantage of components and product line for the organization's Planning Efficiency System. This thesis is divided in three parts. In the first part the theoretical background of components and product platforms is discussed. In the second part an existing system is analyzed and the requirements for the system are defined. The third part consists of the actual architecture design and systematical evaluation process.

This thesis is the starting point for designing systems of SW-Development in future. This thesis also includes information of typical features, challenges, advantages and problems of component-based software development and software product lines.

ALKUSANAT

Tämä diplomityö on toteutettu yhteistyössä SW-Development Oy:n kanssa. SW-Development on Tampereella toimiva ohjelmistoyritys, joka tarjoaa valmistavalle teollisuudelle ohjelmistoratkaisuja sekä konsultointipalveluita. Työn tavoitteena on laatia komponentteja ja tuoterunkoa hyödyntävä arkkitehtuurisuunnitelma SW-Developmentin tuotannonsuunnittelujärjestelmälle.

Kiitän SW-Developmentin hallituksen puheenjohtaja Sivert Westergårdia sekä toimitusjohtaja Jan-Erik Westergårdia yhteistyömahdollisuudesta, tiloista ja työvälineistä. Kiitän työn tarkastajaa, professori Kai Koskimiestä, ja työn ohjaajaa, Antti Halosta, ammattitaitoisesta avusta. Kiitän SW-Developmentin työntekijää, Mikko Kunnaria, avusta ATAM-skenaarioiden laadinnassa sekä järjestelmien analysoinnissa. Erityiskiitos vanhemmilleni, Hannalle ja Pekalle, pitkäjänteisestä ja vankkumattomasta tukemisesta ja avusta niin opiskeluissa kuin elämässäkin.

Tampereella 25.11.2011

Juha-Pekka Arimaa

SISÄLLYS

TERMIT JA NIIDEN MÄÄRITELMÄT

CBD	Component-Based Development, komponenttipohjainen kehitys
CBSE	Component-Based Software Engineering, komponenttipohjainen ohjelmistotuotanto
CLR	Common Language Runtime, .NET Frameworkin ajoaikainen ympäristö
CORBA	Common Object Request Broker Architecture, Object Management Groupin komponenttimalli
COTS	Commercial off-the-shelf, periaate, jossa tuote ostetaan sellaisenaan ilman konfigurointia
DLL	Dynamic Link Library, Windowsin jaettu ajoaikainen kirjasto
ERP	Enterprise Resource Planning, yleisnimitys toiminnanohjausjärjestelmille
JIT-kääntäminen	Just-in-time, menetelmä ohjelman ajoaikaisen tehokkuuden parantamiseen
Komponentti	Itsenäinen ohjelmiston osa
LINQ	Language Integrated Query, .NET Frameworkin datakyselykomponentti
Middleware	Välikerros ohjelmistokomponenttien liittämiseksi
MOTS	Modified off-the-shelf, periaate, jossa tuote konfiguroidaan ostajalle
MVC	Model-View-Control, eräs arkkitehtuurimalli
MVP	Model-View-Presenter, MVC-mallista johdettu arkkitehtuurimalli

PES	Planning Efficiency System, SW-Developmentin rakentama tuotannonsuunnittelujärjestelmä
Suunnittelumalli	Yleinen kuvaus ratkaisusta johonkin tavalliseen ohjelmistokehityksen ongelmaan
SQL	Structured Query Language, standardoitu kyselykieli relaatiotietokantoja varten.
Tuoterunko	Tuoteperheen käyttämä ohjelmistoalusta yhteisellä arkkitehtuurilla
Tuoteperhe	Toiminnaltaan ja rakenteeltaan samankaltaisten ohjelmistojen muodostama joukko
UML	Unified Modeling Language, graafinen mallinnuskieli ohjelmistokehityksen tueksi
XML	Extensible Markup Language, tiedon merkitystä kuvaava merkintäkieli

1 JOHDANTO

Teollisuusmarkkinoilla vallitsee jatkuva kilpailu ja niukkuus, mikä vaatii teollisuusyritysten toiminnan jatkuvaa kehittämistä. Tällaisessa kilpailutilanteessa yritysten tulee pystyä tehostamaan toimintaansa eri tavoin, kuten tekemällä tuotannosta nopeampaa, edullisempaa tai laadukkaampaa. Lisäksi valmistavan teollisuuden yritysten asiakkaiden tarpeet vaihtelevat, jolloin tuotannon tulee olla asiakaskohtaista. Lisäksi muutoksiin tulee pystyä reagoimaan nopeasti. Toiminnan tehostamiseen on luonnollisesti olemassa useita vaihtoehtoja ja yksi keskeisimpiä ratkaisuja on tuotannon tehokas suunnittelu ja hallinta. Tällöin tuotantoprosessia voidaan johtaa ennakoidusti ja tuotantoa voidaan optimoida, jolloin vältetään turhaa työtä ja resurssienkäyttöä. [1, s. 16-22]

Tuotannon tehostamiseen SW-Development on rakentanut tuotannonsuunnittelujärjestelmän nimeltä Planning Efficiency System eli PES. PES on asiakkaan kanssa määritelty ja asiakaskohtaisesti konfiguroitu järjestelmä, joka mahdollistaa tuotantosuunnitelman laatimisen, optimoinnin ja simuloinnin sekä tulosten ja raporttien käsittelyn. Lisäksi PES voidaan integroida asiakkaan omiin järjestelmiin, jolloin käyttöönotto ei edellytä raskaita järjestelmämuutoksia.

Ohjelmistokehityksen kannalta kuvattu järjestelmä aiheuttaa kuitenkin suuria haasteita. Asiakkaille tulisi voida toimittaa räätälöityjä tuotteita, jotka sisältävät yhteisiä ja eriäviä toimintoja. Jokaisen järjestelmän toteuttaminen erillisenä tuotteena ei ole kustannustehokasta, ja samoja asioita tulee tehtyä uudestaan eri asiakkaille. Lisäksi tuotteiden hallinta vaikeutuu, uusien ominaisuuksien toteuttaminen hidastuu ja työtarve kasvaa jokaisen toteutettavan järjestelmän kohdalla.

Tällöin avainasemaan nousee ohjelmistojen ja niiden osien uudelleenkäytettävyys. Komponenttipohjainen kehitys ja tuoterungot voivat molemmat osaltaan vastata näihin vaatimuksiin. Komponenttipohjaisen kehityksen ideana on rakentaa ohjelmisto valmistaa komponenteista, joita voidaan käyttää uudelleen, ja niiden avulla voidaan myös rakentaa erilaisia tuotevariaatioita [2; 3, luku 19]. Tuoterunko on puolestaan tuoteperheelle toteuttava yhteinen alusta, joka sisältää eri tuotteille yhteisen rakenteen ja toiminnallisuuden [4, luku 7].

Ohjelmistokehityksen muutos kohti komponenttipohjaista kehitystä ja tuoterunkoarkkitehtuureja on kuitenkin pitkäaikainen ja haastava prosessi, jolla on vaikutuksia koko organisaatioon. Komponenttien ja tuoterunkojen määrittely ja ominaisuuksien tunnistaminen on ensimmäinen edellytys käyttöönotolle. Tämän lisäksi komponentit ja tuoterungot vaativat riittävästi resursseja, jotta komponentit ja tuoterungot saadaan tehokkaaseen ja hallittuun uudelleenkäyttöön. Luonnollisesti tekninen toteutus asettaa myös omat vaikeudet.

Tämän työn tavoitteena on luoda arkkitehtuurisuunnitelma, jonka avulla komponentteja ja tuoterunkoja voidaan hyödyntää SW-Developmentin tuotannonsuunnittelujärjestelmissä. Luvussa 2 tutkitaan ohjelmistoarkkitehtuureja teoreettisella tasolla ja käsiteltäviä asioita ovat ohjelmistokomponentit, arkkitehtuurityylit, komponenttiarkkitehtuurit, tuoterungot ja tuoterunkojen vaikutus organisaatioon. Luvussa 3 tutkitaan olemassa olevan tuotannonsuunnittelujärjestelmän rakennetta ja toimintaa sekä tunnistetaan keskeisiä ongelmakohtia. Olemassa olevan järjestelmän arkkitehtuuri on lähtökohta komponentteja ja tuoterunkoa hyödyntävälle arkkitehtuurisuunnitelmalle. Luvussa 4 määritetään suunniteltavan arkkitehtuurin vaatimukset. Luvussa 5 suunnitellaan varsinainen arkkitehtuuri määriteltyjen vaatimusten perusteella. Luvussa 6 arvioidaan suunniteltu arkkitehtuuri ATAM-prosessin avulla [4, luku 9]. Luvussa 7 luodaan yleiskatsaus työhön, kootaan keskeiset asiat yhteen ja analysoidaan työn tuloksia.

2 OHJELMISTOARKKITEHTUURIT

Komponenttipohjainen ohjelmistokehitys on ohjelmistotuotannon menetelmä, joka tähtää uudelleenkäytettävyyteen sekä ohjelmiston osien selkeään vastuunjakoon. Siinä missä olio-ohjelmointi pyrkii mallintamaan todellista maailmaa luokkarakenteilla, niin komponenttipohjaisessa ohjelmistokehityksessä pyritään luomaan ohjelmiston osia, joita voidaan liittää toisiinsa ja siten rakentaa kokonainen ohjelmisto. Eräs paljon käytetty analogia ohjelmistokomponenttien kohdalla onkin elektroniikan komponentit, joilla voidaan toteuttaa monimutkaisia piirirakenteita [2]. Komponenttien ajatellaan olevan abstraktimpia ja itsenäisempiä kokonaisuuksia kuin oliot ja toisaalta komponenttipohjainen kehitys voi vastata osaltaan niihin haasteisiin, joihin olio-ohjelmointi ei uudelleenkäytön kannalta ole pystynyt. [3, luku 19]

2.1 Ohjelmistokomponentit

Vaikka ajatus komponenttien käytöstä ohjelmistotuotannossa on peräisin 1960-luvulta, niin vasta 2000-luvulla komponentit ovat saaneet merkittävän roolin ohjelmistokehityksessä [5; 6, luku 1]. Nykyisin kolmannet osapuolet myyvät myös valmiita komponentteja MOTS– ja COTS–periaatteilla, niin tässä työssä keskitytään pääasiassa organisaation omaan komponenttikehitykseen.

Ohjelmistokomponentin ideana on, että se voi tarjota jonkin palvelun riippumatta missä komponenttia käytetään tai millä ohjelmointikielellä komponentti on toteutettu. Komponentti siis kapseloi toiminnallisuutensa, mikä tarkoittaa, että komponentti tulee ottaa käyttöön kokonaisuudessaan. Komponentin rajapinta on julkinen, ja kaikki vuorovaikutus tapahtuu tämän rajapinnan kautta. Täten samaa komponenttia voidaan käyttää useissa eri järjestelmissä, ja yksi järjestelmä voi vastaavasti käyttää myös useampaa komponenttia. Keskeisimpiä tavoitteita komponenttien käytössä onkin uudelleenkäytön ja laadun parantaminen.

2.1.1 Komponentin ominaisuuksia

Ohjelmistokomponentille on olemassa lukuisia erilaisia määritelmiä, mutta sen sijaan, että tukeudutaan tiettyihin määritelmiin, niin komponenttien abstraktin luonteen takia on luontevaa tarkastella komponentteja niiden ominaisuuksien valossa [4, s. 53]. Vaikka komponentit voivat olla hyvinkin erilaisia, niille on mahdollista löytää paljon yhteisiä piirteitä, joiden avulla voidaan luoda kuva siitä, mitä komponentilla tarkoitetaan.

Komponentin tavoitteena on usein korkea itsenäisyyden taso, jolloin komponentti ei ole riippuvainen muista komponenteista. Tällöin komponenttia on mahdollista

käyttää eri ohjelmistoissa ilman, että komponenttiin tai ohjelmistoon joudutaan tekemään raskaita muutoksia. Tällöin useita komponentteja on myös mahdollista kehittää toisistaan riippumatta. Jos komponentilla kuitenkin on komponentin ulkopuolisia vaatimuksia, niin ne tulisi ilmaista eksplisiittisesti komponentin määrittelyssä vaadittuina rajapintoina (rajapintoja käsitellään tarkemmin kohdassa 2.3.1).

Komponentti voidaan ottaa käyttöön liittämällä se osaksi lähdekoodia, mutta myös valmiiksi käännettynä. Komponentin käyttöönotossa tavoitteena on, että komponentin hyödyntäjän ei tarvitsisi ymmärtää komponentin toteutusta vaan vuorovaikutus tehtäisiin julkisten rajapintojen kautta.

Komponentille ei ole määritelty erikseen kokorajoituksia. Tärkeämpi ominaisuus kokoon liittyen on komponentin vastuualue: komponentin tulisi vastata selkeästi määritellyn osakokonaisuuden toiminnallisuudesta. Käytännössä tämä tarkoittaa, että komponentti voi olla hyvinkin pieni tai suuri, mutta toisaalta yleensä komponentin tulisi olla yhden henkilön hallittavissa.

Komponentin tulisi olla dokumentoitu, jotta mahdolliset käyttäjät tietävät täyttääkö komponentti asetetut vaatimukset. Määrittelyssä käytetty syntaksi ja semantiikka tulisi olla myös käyttäjän tiedossa.

Komponentin standardointi tarkoittaa, että komponentin tulisi noudattaa jonkin komponenttimallin asettamia vaatimuksia. Nämä standardit voivat liittyä muun muassa rajapintoihin, metatietoon, dokumentaatioon ja käyttöönottoon. Standardointi on myös edellytys komponenttimarkkinoiden toimivuudelle. [3, luku 19; 4, s. 54-55; 7, s. 5]

2.1.2 Komponenttien hyötyjä ja haittoja

Komponenttien keskeinen hyöty on uudelleenkäytettävyys: kun komponentti on toteutettu, niin sitä voidaan käyttää uudestaan eikä samoja asioita tarvitse tehdä uudelleen. Aikaa säästyy useissa eri ohjelmiston elinkaaren vaiheissa, kuten määrittelyssä, suunnittelussa, testauksessa, integroinnissa ja ylläpidossa. Luonnollisesti mitä korkeampi uudelleenkäyttöaste on, sitä enemmän säästetään aikaa ja rahaa. Liiketoiminnan kannalta ajateltuna uudelleenkäyttö mahdollistaa myös useiden erilaisten tuotevariaatioiden tarjoamisen. Mikäli itsenäisiä komponentteja voidaan liittää vaivattomasti osaksi järjestelmää, niin asiakkaille on helppo tarjota lisää toiminnallisuutta käytettyyn ohjelmistoon. Vastaavasti tämä helpottaa myös konfigurointia asiakkaan näkökulmasta, sillä komponenttien avulla asiakas voi ostaa vain tarvitsemansa toiminnallisuudet. Myöhemmin komponentteja on myös mahdollista lisätä tai poistaa juuri komponenttien itsenäisyyden ansiosta.

Mikäli komponentteja käytetään uudelleen, tulee ne myös testatuksi useaan kertaan. Tämä johtaa luonnollisesti ohjelmiston parempaan laatuun, sillä komponenteissa esiintyviä ohjelmistovirheitä on komponentin elinkaaren aikana ehditty korjata useaan kertaan. Lisäksi komponentit kehittyvät myös käytettävyyden kannalta, sillä kun komponentin käyttökokemuksia on kartoitettu, niin komponenttia on voitu kehittää käyttäjäystävällisemmäksi.

Kuten kohdassa 2.1.1 mainittiin, komponentille ei ole olemassa yksikäsitteisiä kokorajoituksia, mutta sen sijaan komponentin tulisi olla yhden henkilön hallittavissa. Komponenttien tekeminen on siis luonteva tapa jakaa tehtäviä tuotantoryhmän jäsenille. Vastaavasti ohjelmiston pilkkominen komponentteihin jakaa ohjelmiston toiminnallisuutta usein sopiviin osiin, mikä auttaa koko ohjelmiston hallinnassa. [4, luku 3]

Komponenteilla on siis useita hyötyjä, mutta siitä huolimatta komponenttien käyttö ei ole ainakaan vielä niin kutsuttu hopealuoti [8, s. 4; 9]. Komponenttien käyttöön liittyy myös haittapuolia. Laatu ja luotettavuus ovat keskeisiä ongelmakohtia, kun uutta komponenttia otetaan käyttöön. Vaikka komponenttien käyttö pidemmällä aikavälillä usein parantaakin laatua, niin käyttöönottovaiheessa ei voida aina olla täysin varmoja komponentin toimivuudesta. Komponentin itsenäisyyden ja kapseloinnin takia ei myöskään ole tarkoituksenmukaista, että lähdekoodi tutkitaan tarkasti läpi ennen käyttöönottoa. Huomioitavaa on myös, että lähdekoodi ei välttämättä aina ole edes saatavilla ja dokumentointikin voi olla puutteellinen. [3, luku 19]

Komponenttien kanssa saattaa joutua myös tekemään vaihtokauppaa (trade-off) ominaisuuksien kesken. Ohjelmisto saattaa sisältää jonkin selkeän kokonaisuuden, joka olisi hyvä toteuttaa komponentilla, mutta juuri sopivaa komponenttia ei välttämättä ole saatavilla. Sen sijaan käytössä saattaa olla lähestulkoon sopiva komponentti, joka ei kuitenkaan täysin vastaa haluttuihin ominaisuuksiin. Tällöin komponentin voi ottaa käyttöön ja toteuttaa puuttuvat ominaisuudet jollain muulla tapaa, mutta silloin komponentin vastuualue ja itsenäisyys sekä ohjelmiston rakenne saattavat kärsiä. Toisaalta tilalle voi toteuttaa uuden komponentin tai mahdollisesti muokata olemassa olevaa komponenttia sopivampaan muotoon, mutta tällöin taas menetetään komponentin tuomia etuja muun muassa uudelleenkäytön suhteen. Tosin on myös tärkeää ottaa huomioon todellinen tarve komponentille: jos komponentilla ei ole merkittävää uudelleenkäyttöarvoa, saattaa komponentin toteuttaminen olla myös kalliimpaa. [3, luku 19]

Yleiskäyttöisen komponentin tekeminen ei myöskään aina ole helppoa, sillä ohjelmistot ovat usein hyvin laajoja ja monimutkaisia. Tämän takia on hyvin vaikeaa ennustaa kuinka paljon ja minkälaista komponenttia tullaan käyttämään tulevaisuudessa. Ohjelmistot ja komponentit myös kehittyvät jatkuvasti, mikä tekee suunnittelutyöstä entistä vaikeampaa. Komponenttien evoluution ja määrän takia komponenttien hallinta edellyttää myös lisätyötä. Tyypillisesti komponenteille toteutetaan jonkinlainen komponenttikirjasto, mutta komponenttikirjaston käyttö ja hallinta vaativat luonnollisesti resursseja.

2.1.3 Suunnitteluohjeet

Kohdassa 2.1.1 määriteltiin komponentille tyypillisiä ominaisuuksia. Näistä ominaisuuksista saadaan osaltaan johdettua suunnitteluohjeita komponenteille. Komponentteja voidaan ajatella myös pieninä ohjelmina, joten ohjelmistokehityksen yleisemmätkin suunnitteluohjeet ovat sovellettavissa myös komponentteihin. Yksi keskeisimmistä suunnitteluohjeista pohjautuu komponentin itsenäisyyteen. Komponenttia suunniteltaes-

sa on siis erityisen tärkeää pitää komponentti riippumattomana ympäristöstä tai vähintäänkin miettiä riippuvuudet hyvin tarkasti. [3, luku 19; 10]

Komponentin kapseloinnin takia rajapinnat ovat merkittävässä asemassa. Rajapintoihin liittyy läheisesti sopimussuunnittelu (design by contract) sekä tarjotut ja vaaditut rajapinnat. Näitä käsitellään tarkemmin kohdassa 2.3.1. Rajapinnat ovat tyypillisesti pysyviä eikä niihin tehdä muutoksia (vaan julkaistaan tavallisesti kokonaan uusi rajapinta), minkä takia rajapinnat tulee suunnitella erityisen huolellisesti. [3, luku 19; 11, luku 8]

Eräs tapa rakentaa ohjelmistoja on luoda koko ohjelmiston runko ja toteuttaa ohjelmiston osia tämän jälkeen (top-down). Komponenttipohjaiselle ohjelmistokehitykselle voidaan ohjelmisto rakentaa komponenteista eli yhdistää pienempiä osia suureksi kokonaisuudeksi (bottom-up). Tämän takia komponentit on hyvä toteuttaa niin, että ne voidaan testata erillään ohjelmistosta Ilman testauksen ottamista huomioon suunnittelussa, testaaminen saattaa olla huomattavasti vaikeampaa [10; 12].

Kuten kohdan 2.1.2 viimeisessä kappaleessa todettiin, komponentin suunnittelu saattaa olla vaikeaa. Tämän takia komponentin vastuualue tulee määritellä tarkasti. Hyvin yleinen komponentti ei välttämättä vastaa käyttäjän tarpeisiin ja toisaalta juuri tietylle ohjelmistolle toteutettu komponentti ei välttämättä sovellu käytettäväksi muissa ohjelmistoissa. Komponenttia suunniteltaessa tuleekin siis pohtia kuinka paljon ja missä ympäristössä komponenttia tullaan käyttämään ja määritellä toiminnallisuus sen mukaan.

Suunnittelumallien käyttö on vaikuttanut ohjelmistokehitykseen erittäin voimakkaasti 1990-luvulta alkaen. Suunnittelumalli on ohjelmistokehittäjien kokemusten kautta kehittynyt kuvaus jonkin ongelman ratkaisemiseksi. Ohjelmistotekniikassa monien ongelmien luonne on samanlainen, jolloin näihin löytyy samankaltaisia ratkaisuja ja hyväksi todetut ratkaisut on yleistetty suunnittelumalleiksi. Komponenttien sisäisessä toteutuksessa ja ulkoisessa vuorovaikutuksessa on hyvä pitää mielessä erilaisten suunnittelumallien käyttö. Suunnittelumalleja käsitellään tarkemmin kohdassa 2.2.3. [4, luku 5]

Komponenttien suunnitteluohjeita voidaan lisäksi tiivistää SOLID-lyhenteeseen [13, s.135]. Lyhenne muodostuu seuraavasti:

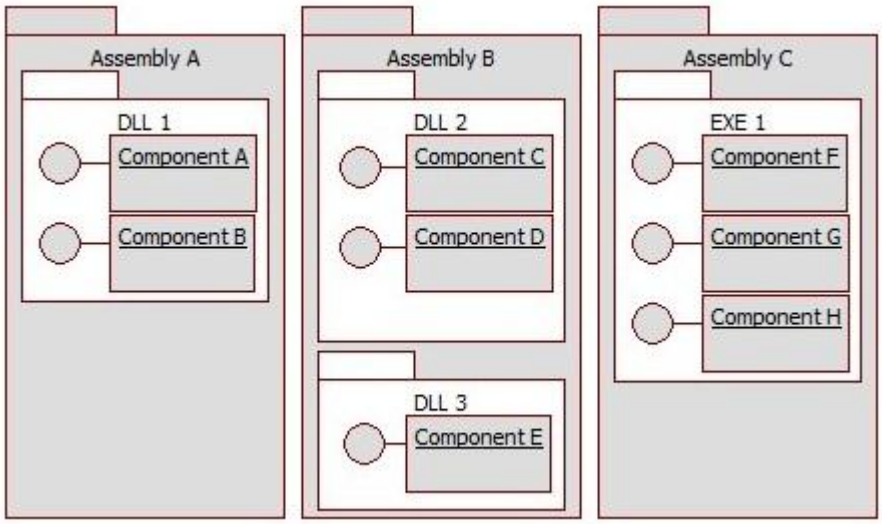
- Single responsibility principle eli yhdellä komponentilla on vain yksi vastuualue.
- Open/closed principle eli komponentin tulee olla laajennettavissa ilman modifointia.
- Liskov substitution principle eli datatyyppi tulee olla korvattavissa alityypeillään.
- Interface segregation principle eli rajapintojen tulee olla kutsujakohtaisia ja hienojakoisia ja eri kutsutarpeisiin tulee toteuttaa erillinen rajapinta.
- Dependancy inversion principle eli riippuvuudet tulee korvata abstraktioilla ja abstraktiot eivät saa olla riippuvaisia yksityiskohdista.

2.1.4 Komponenttiteknologiat

Komponenttiteknologia (myös komponenttimalli) määrittää joukon standardeja komponentin toteutukselle, dokumentoinnille ja käyttöönotolle. Keskeinen tavoite komponenttimallin käytössä on varmistaa komponenttien yhteensopivuus, mutta myös opastaa ohjelmoijia komponenttien kehittämisessä ja käyttöönotossa. Näiden lisäksi komponenttimalliin kuuluu komponenttialusta, joka toimii osaltaan komponenttien käyttöympäristönä. Komponenttialustan merkittävimmät palvelut liittyvät komponenttien pakkaamiseen, tietoturvaan, tiedonsiirtoon ja hajautukseen. On olemassa useita komponenttimalleja, mutta tärkeimpiä ovat OMG:n (Object Management Group) CORBA-malli (Common Object Request Broker Architecture), Oraclen (aikaisemmin Sun Microsystems) JavaBean-malli sekä Microsoftin COM- (Component Object Model) ja .NET-mallit. Tässä työssä keskitytään ainoastaan .NET-malliin, koska se on käsiteltävien järjestelmien kannalta oleellisin. [3, luku 19; 14, s. 22-24]

Vaikka .NET-malliin liittyy hyvin paljon erilaisia ominaisuuksia ja menetelmiä, niin kyseisestä mallista voidaan silti löytää keskeisimmät peruskäsitteet, kuten CLR (Common Language Runtime), koonnit (assemblies), metatiedot ja binääriyhteensopivuus. CLR on ajoaikainen ympäristö, joka tarjoaa yhteisen kontekstin kaikille .NET-komponenteille riippumatta käytetystä kielestä huolehtien myös muistinhallinnasta, jolloin .NET-komponenttimalli on periaatteessa ohjelmointikielestä riippumaton (joskin edellytyksenä luonnollisesti on, että käytettävä kieli on CLR-yhteensopiva, kuten esimerkiksi C#, Visual Basic .NET, Managed C++ ja J#).

Koonnit on kehitetty helpottamaan komponenttien hallintaa lukuisten DLL-tiedostojen (Dynamic Link Library) sijaan (useiden DLL-tiedostojen keskinäinen käyttö sai hankaluutensa takia nimen ”DLL Hell”). Koonti pyrkii ratkaisemaan ongelman nimensä mukaisesti kokoamalla joukon tiedostoja yhdeksi loogiseksi kokonaisuudeksi helpottaen versiointia sekä parantaen jakamis- ja turvallisuusominaisuuksia [kuva 2.1]. Koontia onkin luonnehdittu loogiseksi kirjastoksi, sillä se on metatiedosto, joka voi si-



Kuva 2.1. *Koonnit loogisina yksiköinä. [15]*

sältää useamman tiedoston. Metatiedot liittyvätkin keskeisesti koonteihin, sillä metatiedoilla kuvataan koonnin tyypit, nimiavaruudet, metodit, parametrit, näkyvyydet sekä muut vastaavat asiat. Metatietojen avulla kääntäjä pystyy luomaan koonnista fyysisen tiedoston.

Binääriyhteensopivuus toteutetaan käyttämällä myös metadataa ajoaikaisessa JIT-käännöksessä (just-in-time). Binääriyhteensopivuuden keskeisin hyöty on, että jokainen luokka on binäärikomponentti. Tällöin ei tarvita erilaisia ohjelmistokehyksiä, koska .NET tukee komponentteja luonnostaan. Hyödyt eivät tosin rajoitu vain rajapintoihin, sillä myös metodit ja tyypit ovat binääriyhteensopivia, jolloin esimerkiksi metodin lisääminen on vaivatonta. [15, luku 2]

2.1.5 Uudelleenkäytettävyys

Komponenttien uudelleenkäytettävyyttä on jo käsitelty muun muassa kohdassa 2.1.2 ja komponentin suunnitteluohjeita kohdassa 2.1.3, mutta koska uudelleenkäytettävyys on tässä työssä käsiteltävän tuotannonsuunnittelujärjestelmän kannalta erittäin keskeinen ominaisuus, perehdytään uudelleenkäytettävyyteen vielä hieman syvemmin. Uudelleenkäytettävyyden suhteen komponentin tärkeimpiä piirteitä ovat komponentin itsenäisyys, tietotyyppien määrittelemättömyys, toteutuksen piilottaminen, kieliriippumattomuus ja standardointi.

Yleisesti ottaen komponentit tulee luoda yleiskäyttöisiksi eikä kehittää niitä vain jotain tiettyä sovellusta varten. Komponentti on todennäköisemmin uudelleenkäytettävä mikäli se tietyn ohjelmiston sijaan sidotaan johonkin yleisempään ja pysyvämpään abstraktioon (domain abstraction). Sairaalaan liittyvä yleisempi abstraktio voisi olla esimerkiksi potilas, koska se liittyy fundamentaalisesti sairaalaan eikä yhteys sairaalaan ja potilaan välillä luultavasti muutu kovinkaan nopeasti. Yleiskäyttöisyyden suhteen saattaa tosin joutua tekemään vaihtokauppaa soveltuvuuden kanssa: mitä yleisempi rajapinta on, sitä monimutkaisempi ja vähemmän sovellettava komponentti on.

Uudelleenkäytettävyydelle voidaan myös löytää hyviksi havaittuja toteutusmenetelmiä. Komponentin tulisi toteutuksensa lisäksi piilottaa myös tilansa itsenäisyyden ylläpitämiseksi. Poikkeuksien tulisi olla osa komponentin rajapintaa eikä poikkeuksia tulisi käsitellä komponentin toteutuksessa, koska eri sovelluksilla on oletettavasti omat tapansa poikkeustenkäsittelyyn. Komponentin sovittamista ohjelmaa varten on hyvä tehdä erillinen konfiguraatiorajapinta. Jos komponentti käyttää muita komponentteja, niin mahdollisuuksien mukaan ne tulisi integroida kokonaisuudessaan toteuttavaan komponenttiin riippuvuuksien vähentämiseksi. Nimeämisessä kannattaa olla johdonmukainen ja suosia yleisluontoisia nimiä etenkin rajapintojen suhteen.

Uudelleenkäytettävyydessä tulee huomioida myös tehokkuus- ja kustannusvaikutukset. Yleiset komponentit ovat tyypillisesti hitaampia ja suurempia kuin tietylle sovellukselle tehdyt spesifit toteutukset – tässäkin asiassa tulee pohtia mitkä ovat tärkeimpiä laatuominaisuuksia. Uudelleenkäytettävä komponentti on myös hitaampaa ja kalliimpaa kehittää, minkä takia siihen kuluvat resurssit olisi hyvä allokoida organisaatiolle eikä yksittäiselle projektille. [3, luku 19]

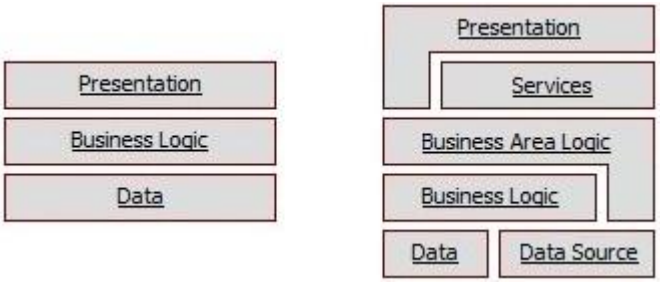
2.2 Arkkitehtuurityylit ja suunnittelumallit

Jotta voidaan syventyä tarkemmin komponenttipohjaiseen arkkitehtuuriin, on syytä käsitellä ohjelmistoarkkitehtuureja yleisellä tasolla. Komponenttien käyttöä voidaan pitää eräänä lähtökohtana ohjelmistoarkkitehtuurille ja komponenttiarkkitehtuureihin luonnollisesti liittyy siten myös arkkitehtuurityylit ja suunnittelumallit.

2.2.1 Arkkitehtuurityylit

Arkkitehtuurityyli (tunnetaan myös nimellä arkkitehtuurimalli) on joukko periaatteita, joiden avulla järjestelmän arkkitehtuuria voidaan hahmottaa ja luokitella korkealla tasolla. Lisäksi arkkitehtuurityyli tarjoaa yhteisen kielen ja mahdollistaa korkean tason keskustelua ilman, että joudutaan puuttumaan yksityiskohtiin. Arkkitehtuurityylejä on olemassa useita, joista tunnetuimpia ovat kerrosarkkitehtuuri, tietovuoarkkitehtuuri, asiakas/palvelin–arkkitehtuuri, palvelukeskeinen arkkitehtuuri (SOA eli service-oriented architecture), Model-View-Controller–arkkitehtuuri (MVC) ja oliokeskeinen arkkitehtuuri. On hyvin tavallista, että arkkitehtuurityylejä yhdistellään eli voidaan esimerkiksi toteuttaa kerrospohjainen järjestelmä komponentteja käyttäen siten, että näkymätaso noudattaa MVC-mallia, ja lisäksi järjestelmä voi kommunikoida asiakas/palvelin–mallin mukaisesti jonkin ulkoisen systeemin kanssa. Kaikkia tyylejä ei käydä läpi vaan keskitytään tutkittavan järjestelmän kannalta keskeisimpiin tyyleihin. [4, s. 125; 13, s. 20]

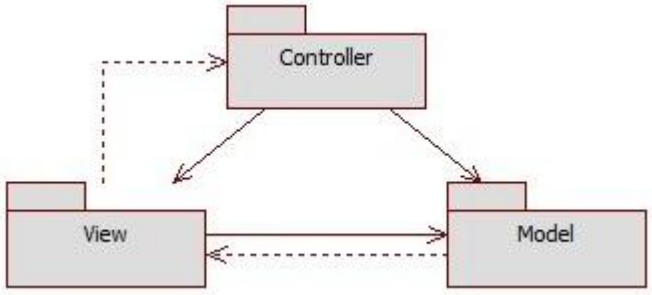
Kerrosarkkitehtuurissa [kuva 2.2] yksi osa-alue vastaa yhtä kerrosta ja yksittäinen kerros kommunikoi ylemmän ja/tai alemman tason kerroksen kanssa. Kolmikerrosarkkitehtuurissa käyttöliittymä on sijoitettu Presentation-kerrokselle, toimintalogiikka Business Logic –kerrokselle ja datan käsittely Data-kerrokselle. Kerrosarkkitehtuurin hyvä puoli on sen selkeys, abstrahointi, kapselointi, uudelleenkäytettävyys, tiiviit kerrokset ja vähäiset riippuvuudet, mutta haittapuolena on usein heikompi tehokkuus, sillä



Kuva 2.2. *Yksinkertaistettu kerrosarkkitehtuuri, jossa ei tehdä ohituksia (vasemmalla) ja monimutkaisempi kerrosarkkitehtuuri ohituksien kanssa (oikealla).*

kutsuja saatetaan joutua tekemään useita eri kerrosten välillä. Kutsujen määrää voi luonnollisesti vähentää tekemällä ohituksia hyppäämällä jonkin kerroksen yli, mutta mikäli ohituksia tehdään paljon, niin kerrosarkkitehtuurin idea luonnollisesti kärsii. Huomattavaa on, että kerrosarkkitehtuureissa puhuttaessa on hyvä erottaa tarkoitetaanko kerroksilla loogisesti vai fyysisesti eri kerroksia (englannin kielessä nämä on erotettu sanoilla tier eli fyysinen kerros ja layer eli looginen kerros).

MVC-mallin keskeinen idea on, että järjestelmän käyttöliittymä voidaan erottaa sovelluslogiikasta ja järjestelmän käyttämästä datasta. Tällöin erilaisia käyttöliittymänäkymiä voidaan helposti toteuttaa. MVC-mallissa Model-osan vastuulla on huolehtia järjestelmän datasta tai tilasta. View-osan vastuulla on puolestaan erilaisten käyttöliittymänäkymien käsittely, ja Controller-osa toimii mallin ja näkymän välissä huolehtien järjestelmän toimintalogiikasta. Puhtaan MVC-mallin perusidea on esitetty kuvassa 2.3. Kuvassa 2.3 kiinteät nuolet tarkoittavat suoraa assosisaatiota osien välillä ja katkoviivalla merkityt nuolet puolestaan epäsuoraa assosisaatiota. Käytännössä epäsuoran assosisaation voi toteuttaa esimerkiksi Tarkkailija-suunnittelumallia (Observer) käyttäen [2.2.2]. [4, s. 142-144; 13, s. 19-35]



Kuva 2.3. *MVC-mallin perusidea.*

2.2.2 Suunnittelumallit

Suunnittelumalli on käytännön kokemusten kautta hyväksi osoittautunut yleinen kuvaus ratkaisusta ongelmaan, jolla on tapana toistua erilaisissa yhteyksissä. Käytännössä tietynlaiset ongelmat nousevat esiin eri vaiheissa, ja ajan myötä niiden ratkomiseen on muodostunut hyväksi havaittuja ratkaisuja, joita voidaan käyttää uudelleen. Sen sijaan, että itse toteuttaa omat ratkaisut jokaiseen ongelmaan, voi käyttää avukseen valmista suunnittelumallia, mikä oletettavasti helpottaa suunnittelua ja parantaa laatua. Suunnittelumalli kuvaa ratkaisun yleisellä tasolla (eikä sido ratkaisua esimerkiksi tiettyyn ohjelmointikieleen), minkä takia ne ovatkin saavuttaneet suuren suosion ja erilaisia suunnittelumalleja on kehitetty lukuisia määriä. Tässä työssä ei kuitenkaan käsitellä eri suunnittelumalleja vaan perehdytään suunnittelumallin ideaan (eräs esimerkki suunnittelumallista käsitellään kohdassa 2.3.2). Vaikka suunnittelumallit ovat alun perin keskittyneet olio-ohjelmointiin, niin ne soveltuvat hyvin myös komponenttipohjaisen ohjelmistokehityksen tarpeisiin.

Suunnittelumallilla on neljä keskeistä osaa: nimi, ongelma, ratkaisu ja seuraamukset. Nimi kuvaa suunnittelumallin ongelmaa, ratkaisua ja seuraamuksia lyhyesti. Hyvän nimen avulla voidaan monimutkaiset suunnittelurakenteet käsittää yksiselitteisellä nimellä ja kommunikoida abstrakteilla käsitteillä. Ongelma kuvaa, milloin kyseistä suunnittelumallia tulisi soveltaa tai mihin ympäristöön ratkaisu sopii. Ratkaisu kuvaa mallin eri osien suunnittelun, osien suhteet ja vastuut sekä osien yhteistoiminnan. Seuraamukset kertovat mitä hyötyjä ja haittoja mallin käytöstä seuraa – parempi ylläpidettävyys saattaa esimerkiksi heikentää tehokkuutta. [16, luku 1]

Suunnittelumallien käytön myötä myös antisuunnittelumallit ovat yleistyneet. Antisuunnittelumallit ovat vastaavasti yleisiä ratkaisuja samantyyppisiin ongelmiin, mutta erona on, että antisuunnittelumalli esittelee huonon ratkaisun ongelmaan. Antisuunnittelumallin hyöty kuitenkin on, että sen avulla voidaan tunnistaa huonoja ratkaisuja ja mahdollisesti korvata huono ratkaisu paremmalla (esimerkiksi jollain suunnittelumallilla). Kuten suunnittelumalleja, niin antisuunnittelumallejakin on löydetty ja kehitetty lukuisia. [4, s. 107-109]

2.3 Komponenttipohjainen arkkitehtuuri

Komponenttipohjaisella arkkitehtuurilla tarkoitetaan arkkitehtuuria, joka suunnitellaan valmiita komponentteja yhdistelemällä ja hyväksikäyttäen. Tällöin keskeistä on huomioida komponenttien edellyttämät yhteensopivuusvaatimukset, jotta komponentteja voidaan todella liittää osaksi järjestelmää. Komponenttien liittäminen edellyttää puolestaan vuorovaikutusta komponenttien ja järjestelmän välillä, jolloin muun muassa rajapinnat ovat merkittävässä asemassa. Näiden lisäksi tulee huomioida myös komponenttien riippuvuudet järjestelmää suunniteltaessa, jolloin suunnittelumalleja voidaan käyttää hyväksi.

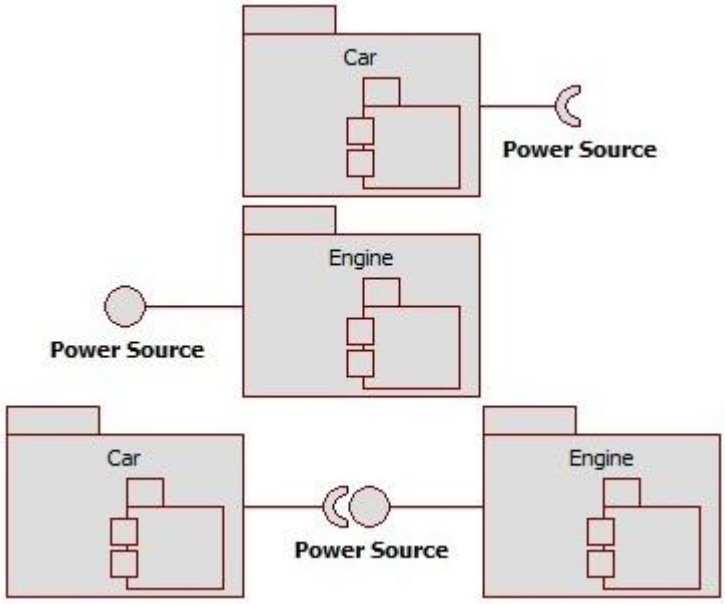
2.3.1 Rajapinnat

Rajapintoja on käsitelty hieman jo aliluvussa 2.1, mutta komponenttipohjaisessa arkkitehtuurissa rajapinnat ovat erittäin keskeisessä asemassa, joten niihin perehdytään vielä tarkemmin. Suppeammin ilmaistuna rajapinta kuvaa kuinka komponentti otetaan käyttöön, mutta laajemmin ajateltuna rajapinta kertoo kaiken, mitä käyttäjän komponentista tulisi tietää (kuten komponentin laadullisia ominaisuuksia). Rajapintojen keskeinen asema johtuu siitä, että niiden avulla komponentit kommunikoivat järjestelmän muiden osien kanssa, mutta rajapinta määrittää myös komponentin laatupiirteitä, joita ovat muun muassa testattavuus ja joustavuus. Usein on myös tavoitteena, että komponentin rajapintaa ei muuteta vaikka komponentin sisäistä toteutusta muutettaisiinkin. Tavallisesti julkaistaan kokonaan uusi rajapinta, mikä korostaa entisestään rajapinnan merkitystä. [4, s. 57; 17; 18, s. 4]

Rajapinnan määrittämille palveluille voidaan antaa tulo- ja jättöehtoja (toisinaan kutsutaan myös esi- ja jälkiehdoiksi). Nämä tarkoittavat eräänlaista sopimusta palvelun käyttäjän ja toteuttajan välillä: käyttäjä lupaa käyttää palvelua tuloehtojen mukaisesti ja palvelun toteuttaja lupaa muuttaa olosuhteet jättöehtojen mukaisiksi. Tällaista menetelmää kutsutaankin sopimussuunnitteluksi, ja menetelmän avulla voidaan määrittää rajapinnan käyttöä koskevia vaatimuksia ja vastuita. Voidaan ajatella, että mikäli tuloehtoja ei noudateta, niin ei palvelun tarvitse toimiakaan oikein. Nykyään tosin suositaan turvallista tai puolustavaa ohjelmointia (defensive programming), jossa varaudutaan muun muassa tuloehtojen rikkomiseen [19, luku 8]. Tulo- ja jättöehtojen toteuttaminen voi olla vain sanallinen selitys dokumentaatiossa, mutta toisaalta myös lähdekoodiin upotettu tarkastelu, joka herättää esimerkiksi poikkeuksen, mikäli tuloehtoja rikotaan. [4, s. 62; 10, s.197-202]

Myös invariantteja voidaan käyttää tulo- ja jättöehtojen lisäksi. Invariantti tarkoittaa lauseketta, jonka arvo ei muutu suorituksen aikana. Erittäin yksinkertainen invariantti voisi olla, että tietorakenteen alkioiden määrä ei saa olla pienempi kuin nolla. Invariantteja voi määritellä palveluille, luokille tai kokonaisille komponenteille esimerkiksi ohjelman tilaa koskien. Invarianttien suunnittelu ja käyttö voi osaltaan helpottaa komponentin testausta ja ylläpitoa. [4, s. 62-65]

Rajapintoihin liittyy kiinteästi myös käsitteet tarjottu ja vaadittu rajapinta [kuva 2.4]. Nämä käsitteet tarkoittavat, että komponentti voi tarjota omia palveluitaan, jolloin



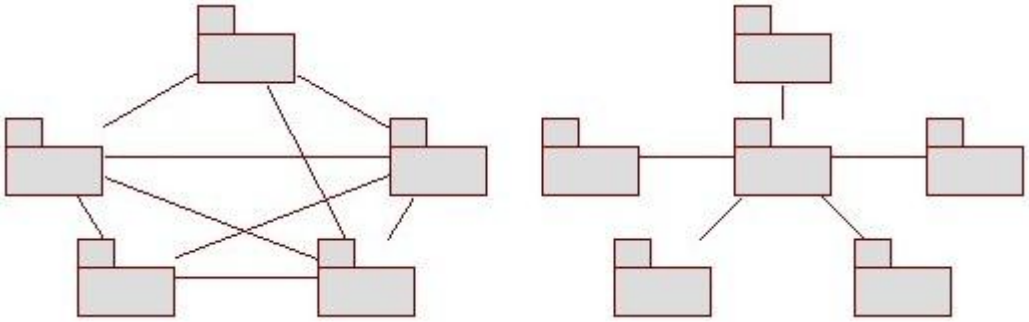
Kuva 2.4. *Tarjotut ja vaaditut rajapinnat. [4]*

se muodostaa tarjotun rajapinnan. Vastaavasti komponentin toiminta voi edellyttää, että sen käytössä on jokin rajapinta, joka toteuttaa ne palvelut, joita komponentti tarvitsee. Tällöin yhden komponentin tarjoama rajapinta voi olla jonkin toisen komponentin vaadittu rajapinta. [4, luku 3; 7, s. 9]

2.3.2 Vuorovaikutus ja välikerros

Komponentit toimivat rajapintojensa kautta ja kuten kohdassa 2.3.1 mainittiin, komponenteilla voi olla sekä tarjottu että vaadittu rajapinta. Näin ollen komponentit luovat riippuvuussuhteita toisiinsa, ja komponentit vuorovaikuttavat keskenään. Yleisperiaate riippuvuussuhteille on ”high cohesion and loose coupling” eli komponentin tulisi olla yhtenäinen eikä sillä saisi olla vahvoja riippuvuuksia [17]. Riippuvuuksilla ja vuorovaikutuksilla on merkittävä rooli järjestelmän laadullisia ominaisuuksia – kuten ylläpidettävyyttä ja uudelleenkäytettävyyttä – tarkasteltaessa, minkä takia komponenttien riippuvuussuhteille on kehitetty useita ratkaisumalleja. Tarkastellaan yksinkertaista kuvitteellista ohjelmistoa, joka koostuu viidestä komponentista. Komponentit käyttävät toistensa palveluita ja ovat siten riippuvaisia toisistaan. Tämä on kuitenkin ristiriitainen yllämainitun periaatteen suhteen, mikä aiheuttaa luultavasti muun muassa ongelmia muunneltavuuden suhteen. Komponenttien ylimääräisten riippuvuussuhteiden poistamisessa on tyypillistä luoda uusia rajapintoja ja epäsuoruuksia komponenttien välille. Eräs ratkaisu

tämän ongelman poistamiseksi onkin luoda uusi välittäjäkomponentti, jonka avulla monta-moneen-riippuvuudet voidaan muuttaa yksi-moneen-riippuvuuksiksi [kuva 2.5]. Tämä tuo osaltaan esiin komponenttien vuorovaikutuksen merkitystä uudelleenkäytettävyydessä – ei siis riitä, että komponentti itsessään on uudelleenkäytettävä. Vastaavia suunnittelumalleja on lukuisia ja niitä käsiteltiin tarkemmin kohdassa 2.2.2. [4, luku 4]



Kuva 2.5. *Välittäjäkomponentin käyttö. [4]*

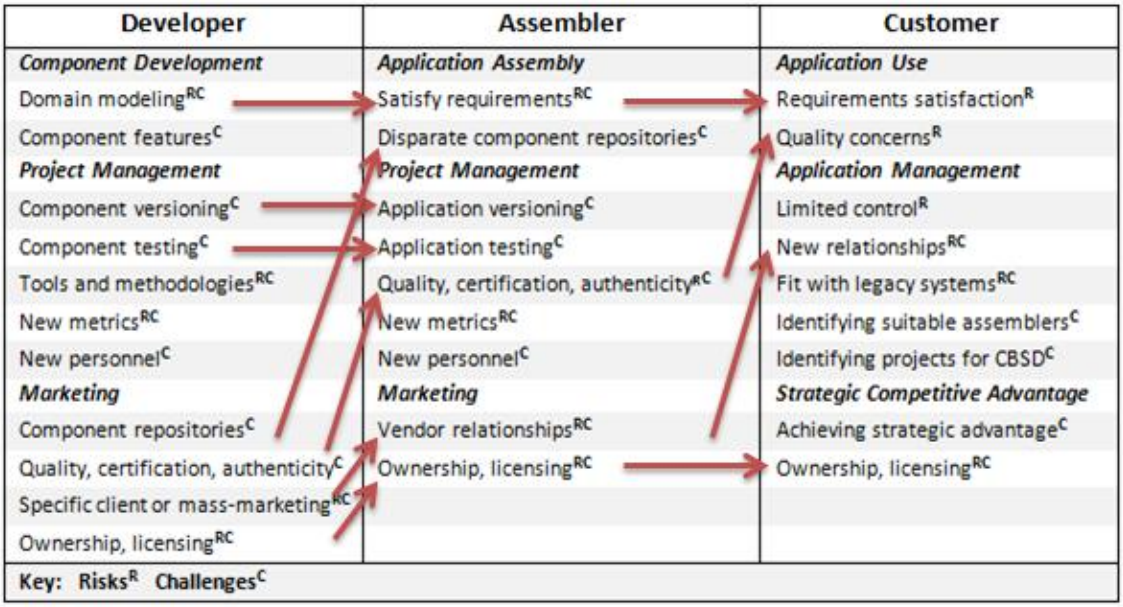
Laajasti ajateltuna välikerros (toisinaan myös väliohjelmisto) tarkoittaa mitä tahansa ohjelmistoa, joka mahdollistaa muiden ohjelmistojen tai komponenttien vuorovaikutuksen [20]. Välikerrosta on myös kuvattu liimaksi komponenttien välille. Sen lisäksi, että komponentit kommunikoivat keskenään, komponentit saattavat kommunikoida myös esimerkiksi tietokannan tai käyttöjärjestelmän kanssa. Tämä kommunikointi ei useinkaan toteudu suoraan vaan erilaisten välikerroksien kautta. Tässä työssä käsiteltävän järjestelmän kannalta merkittävimmän välikerroksen muodostavat .NET Frameworkin lukuisat ominaisuudet ja palvelut. Eräs esimerkki tämän ohjelmistokehyksen välikerroksesta on LINQ (Language Integrated Query), joka mahdollistaa datakyselyjen suorittamisen. LINQ mahdollistaa muun muassa sen, että datakyselyjä voidaan tehdä samantyyppisellä semantiikalla XML-dokumenteista, taulukoista tai relaatiotiekannoista eri ohjelmointikielillä, mikä edistää luonnollisesti muunneltavuutta ja ylläpidettävyyttä. [21, luku 2]

2.3.3 Keskeisiä haasteita

Komponenttipohjaisen arkkitehtuurin käyttö asettaa useita erilaisia haasteita ja riskejä organisaation eri osa-alueille [kuva 2.6]. Komponentin kehittäjät joutuvat tarkastelemaan komponentin laatuominaisuuksia ja toimialan mallinnusta. Projektinhallinnan kannalta komponenttien versiointi ja yksikkötestaus, komponenttimallit ja työkalut saattavat olla ohjelmiston elinkaaren kannalta hyvinkin haastavia tekijöitä – etenkin muutosten alla. On myös tyypillistä, että yhtä komponenttia kehittää yksi henkilö, jolloin henkilöstön hallinta on myös keskeisessä roolissa muun muassa henkilöstön vaihtuvuuden takia. Myös markkinointi- ja myyntiosasto joutuvat kiinnittämään huomioita komponenttien mahdollistamaan tarjoamaan: millaista konfiguraatiota ja toiminnallisuutta voidaan asiakkaalle luvata. Lisäksi lisenssiasiat tulee olla kunnossa, mikäli käytetään kolmannen osapuolen komponentteja.

Järjestelmällä on luonnollisesti asetettu tietyt vaatimukset ja järjestelmän kannalta onkin keskeistä, että täyttävätkö käytetyt komponentit nämä vaatimukset. Myös koko

järjestelmän versionhallinta edellyttää resursseja jo pelkästään komponenttien eri versioiden takia, sillä yhteensopivuus ei aina ole taattua, vaikka rajapinnat olisivatkin muuttumattomia. Jos komponentteja kehitetään erillään järjestelmästä, niin integrointi ja järjestelmätestaus vaikeutuvat, sillä koko järjestelmä ei välttämättä ole aina käytössä



Kuva 2.6. *Keskeisimmät haasteet ja niiden kumuloituvat vaikutukset. [22]*

(etenkin bottom-up-suunnittelua käytettäessä).

Haasteet ja riskit heijastuvat osittain myös asiakkaalle. Tärkeimpiä asioita ovat muun muassa vaatimusten täyttäminen: saadaanko tiettyjä komponentteja käyttämällä haluttu toiminnallisuus laadusta tinkimättä. Lisäksi asiakas saattaa joutua ongelmiin vanhojen järjestelmien ohjelmistopäivitysten kanssa, mikäli ne eivät ole yhteensopivia uusien komponenttien kanssa. Asiakas voi myös joutua ostamaan joidenkin komponenttien käyttöoikeuksia mikäli haluavat käyttää jotain tiettyä järjestelmää. [7, s. xxxi; 22]

2.4 Tuoterungot

Tuoteperhe on samankaltaisten ohjelmien muodostama kokonaisuus ja kyseiset ohjelmat käyttävät yhteistä tuoterunkoa, joka toteuttaa ohjelmien yhteisen rakenteen ja toiminnallisuuden. Yhdestä tuoterungosta voidaan siis rakentaa useita ohjelmistoja, jotka kuuluvat siten samaan tuoteperheeseen. Vastaavasti tuoterunkoarkkitehtuurilla tarkoitetaan tuoterungon arkkitehtuuria, joka osaltaan ohjaa tuoteperheen ohjelmistojen rakentamista. Tuoteperheen ohjelmistot siis liittyvät samaan sovellusalueeseen, ja nämä ohjelmistot jakavat yhteisen arkkitehtuurin, ja ohjelmistoja rakennetaan edelleen komponentteja hyväksikäyttäen. Tuoterunkoja voi myös ymmärtää paremmin käsittelemällä sitä mitä tuoterungot eivät ole. Tuoterungolla ei tarkoiteta pelkästään

- yhden tuotteen kehittämistä käyttämällä koodia uudestaan jostain toisesta ohjelmistosta
- komponenttipohjaista kehitystä

- tuoterunkoarkkitehtuurin muokkaamista uusiokäyttöön
- eri versioiden tekemistä samasta tuotteesta
- joukkoa teknisiä standardeja.

Sen sijaan tuoterunko sitoo yhteen yllämainittuja asioita. [4, luku 7; 23]

2.4.1 Tuoterungon hyödyt ja haitat

Tuoterungosta koituu monia hyötyjä, jotka ovat osaltaan samankaltaisia kuin komponenttien kohdalla. Uudelleenkäyttö on yksi selkeimmistä hyödyistä: kuten komponenteissakin, niin myös tuoterungoissa ohjelmisto tulee testattua useaan kertaan ja ohjelmiston rakentaminen vaatii vähemmän resursseja (kuten aikaa, rahaa ja henkilöstöä). Uudelleenkäytön myötä erilaisten konfigurointien toteuttaminen on myös helpompaa. Tämän lisäksi tuoterungon käyttö voi helpottaa projektien läpivientiä, sillä vastaavanlaisia projekteja on toteutettu jo aikaisemmin – voidaan muun muassa käyttää samantyylistä projektinhallintaa sekä vastaavia prosessimalleja. Tuoterunko jakaa myös tietoa organisaation sisällä, sillä tuoterunko on oletettavasti usean kehittäjän käytössä, jolloin usealla henkilöllä on käsitys ohjelmistojen rakenteesta ja toiminnallisuudesta. Tällöin myös siirtyminen projekteista toiseen on helpompaa, sillä tuoterunko on entuudestaan tuttu.

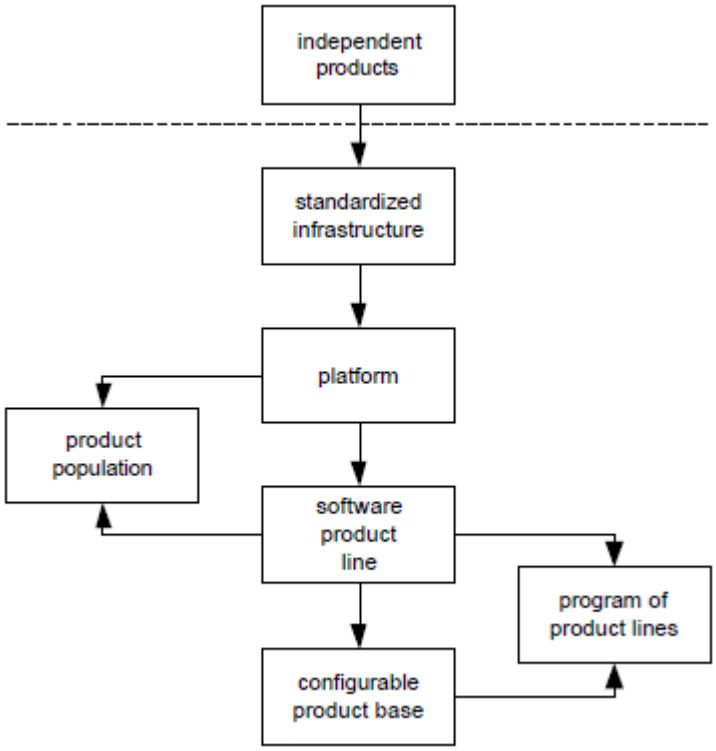
Toisaalta tuoterunko saattaa aiheuttaa myös ongelmia henkilöstön suhteen. Jos tuoterunkoa kehitetään erillään varsinaisten ohjelmistojen rakentamisesta, niin tietotaito saattaa keskittyä liikaa tuoterungon kehittäjille. Tuoterunkojen kehittäjien motivaatiota voi myös laskea se, että nämä kehittäjät eivät pääse toteuttamaan mitään valmista tuotetta. Organisaation kannalta tuoterunko edellyttää suunniteltua kehittämistä, sillä toimiva tuoterunko harvoin syntyy valmiiden, erillisten tuotteiden pohjalta, vaikka jokin tuote valittaisiinkin päätuotteeksi. Tämä kehitys voi aiheuttaa myös ongelmia. Erillinen kehitystyö, josta hyödytään vasta pitkän ajan päästä, ei sovi kovinkaan hyvin kvartaalitalouteen. Lisäksi tämä kehitystyö pitäisi olla ainakin osittain irrallaan projekteista, sillä asiakkaat eivät luultavammin halua maksaa tuoterungosta, ja kehitys voi häiritä myös projektin aikataulua. Useiden tuoteperheen ohjelmistojen hallinta vaatii myös systemaattista toimintaa ja lisäresursseja. Valmiin tuoteperheen ohjelmiston myötä tuoterunko tulee testattua useaan kertaan, mutta varsinaista tuoterunkoa on vaikea testata ilman, että siitä on rakennettu jokin valmis tuote. [4, luku 7; 23]

2.4.2 Tuoterungon rakentaminen

Tuoterungon rakentaminen on pitkäkestoinen prosessi ja tuoterungon rakentaminen voi alkaa erilaisista lähtökohdista riippuen organisaation tilanteesta. Lisäksi tuoterungoille voidaan erottaa erilaisia kypsyysasteita sen mukaan kuinka laajaa uudelleenkäyttö on [kuva 2.7]. Tarkastellaan ensin erilaisia kypsyysasteita ja perehdytään sen jälkeen erilaisiin lähtökohtiin.

Yhdenmukainen infrastruktuuri (standardized infrastucture) on ensimmäinen askel eri tuotteiden yhteisten ominaisuuksien hyödyntämisessä. Yhdenmukainen infrastruktuuri sisältää tyypillisesti sovellettavan käyttöjärjestelmän, tietokantahallinnan ja graafisen käyttöliittymän. Tällä tasolla ei vielä hyödynnetä eri tuotteiden toiminnallisuuden uudelleenkäyttöä.

Alustan (platform) rakentaminen edellyttää vaatimusmäärittelyä ja käsitemallia,



Kuva 2.7. *Tuoterungon kypsyysasteet. [24]*

sillä alusta sisältää kaikille toteutettaville sovelluksille yhteisen toiminnallisuuden. Vaatimusmäärittely ja käsitemalli rajaavat alustan toimialueen ja antavat olennaiset käsitteet. Tämän jälkeen voidaan suunnitella alustan arkkitehtuuri huomioiden variaatiopisteet ja muunneltavuusvaatimukset.

Alustan rakentamisen jälkeen seuraava askel on tuoterunko (software product line), jossa eri sovelluksilla on jaettu yhteistä toiminnallisuutta, mutta kaikkea toiminnallisuutta ei jaeta kaikkien sovellusten kesken. Tällöin muunneltavuuden hallinta edellyttää jo huomattavan paljon resursseja, koska tuoterunko sisältää enemmän variaatiopisteitä.

Konfiguroitava tuote (configurable product base) tarkoittaa, että organisaatio on valmistanut tuotepaketin, joka sisältää kaiken toiminnallisuuden, mutta vain sovittu toiminnallisuus annetaan asiakkaan käyttöön. Toiminnallisuus voidaan konfiguroida automaattisesti esimerkiksi asiakkaan hankkiman lisenssin mukaan. Tämä kypsyystaso edellyttää vakaata toiminta- ja markkina-aluetta, sillä toisistaan paljon poikkeavien tuotteiden valmistaminen ei käytännössä onnistu konfiguroitavalla tuotteella. Tällöin käytettävät resurssit sijoitetaankin koko tuotepaketin rakentamiseen eikä erilaisten tuotteiden kehitykseen.

Tuoterunkojen yhdistelmä (program of product lines) kokoaa yhteen useita tuoterunkoja, jolloin yksittäinen tuoterunko on koko sovelluksen osajärjestelmä. Luonnollisesti tämä soveltuu käytännössä vain erittäin laajoille järjestelmille, joissa ominaisuuksia halutaan lisätä. Muunneltavuuden hallinta on tällöin erittäin suuri haaste ja muunneltavuuteen tuleekin kiinnittää paljon huomiota. Tuotteiden lisääminen (product population) sen sijaan tarkoittaa, että tuoterunkojen ominaisuuksia yhdistelemällä luodaan suurempi määrä erilaisia tuotteita (eikä yksittäisen tuotteen laajentamista). [24]

Tuoterunkoa voi myös lähteä rakentamaan erilaisista lähtökohdista riippuen organisaation tilanteesta. Näistä lähtökohdista voidaan eritellä ainakin neljä erilaista menetelmää:

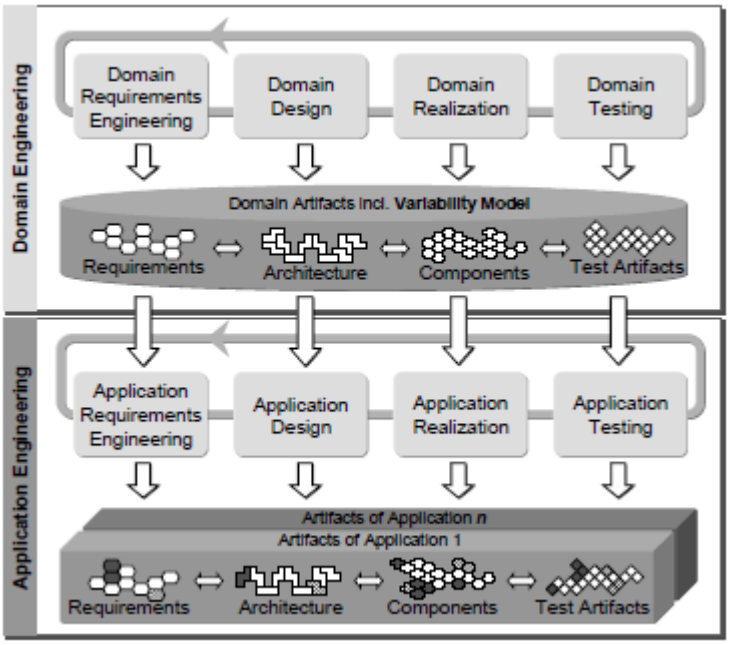
1. Olemassa olevat komponentit: organisaation rakentamat valmiit komponentit otetaan toteutettavan tuoterungon perustaksi.
2. Uusi tuoterunko inkrementaalisesti: uutta tuoterunkoa aletaan rakentaa vähitellen siten, että ensimmäisessä tuoterungossa on vähän ominaisuuksia, mutta ajan myötä ominaisuuksien määrää kasvatetaan.
3. Uusi tuoterunko valmiiksi: uusi tuoterunko rakennetaan kokonaisuudessaan ja tätä tuoterunkoa käytetään jatkossa kehitettävien tuotteiden runkona.
4. Mallituote: rakennetaan ensin yksi tuoterunkoajattelua noudattava tuote ja kehitetään kyseistä tuotetta täysimittaiseksi tuoterungoksi. [4, luku 7; 14, s. 9-19]

2.4.3 Muunneltavuus

Muunneltavuus ja muunneltavuuden hallinta ovat tuoterunkojen keskeisimpiä ongelmakohtia. Tuoterungoista voidaan erotella alusta- ja sovelluskehitys [kuva 2.8]. Tuoterungosta tulisi pystyä rakentamaan erilaisia tuotteita tuoteperheeseen ja tuotteiden ominaisuudet luonnollisesti vaihtelevat. Tuoterungon alusta sisältää kaikille tuotteille yhteiset ominaisuudet ja tuoterungon ominaisuuksia voidaan muunnella esimerkiksi komponenteilla. Komponentit voivat olla valinnaisia tai vaihtoehtoisia ja komponentit voivat olla myös yhteisiä eri tuotteissa. Muunneltavuus tulee ottaa huomioon kehityksen kaikissa eri vaiheissa. [4, s. 168-172]

Kuva 2.8 tuo esiin tuoterungon alustan ja tuoterungon avulla rakennettavan tuotteen erittelyn. Molemmille on olemassa omat vaatimukset, suunnitteluratkaisut ja testaustoimenpiteet, vaikka sovellus rakennetaankin yhteisen alustan päälle. Niin alustassa kuin tuoterungossa tulee ensin huomioida vaatimukset ja määritellä muunneltavuuspiirteet vaatimusten mukaan. Määrittelyvaiheessa tulee huomioida eri toiminnallisuusvaatimukset: on mahdollista, että tuotteilla on paljonkin yhteistä toiminnallisuutta tai toisaalta myös täysin tuotekohtainen toiminnallisuus on mahdollinen. Määrittelyn avuksi voi luoda muun muassa piirremallin, joka voi esimerkiksi olla UML:n avulla toteutettava kaavio eri muunneltavuusvaihtoehdoista.

Määrittelyn jälkeen tulee suunnitella ja toteuttaa määrittelyn mukainen toiminnallisuus. Käytettävät teknologiat voivat asettaa erilaisia ehtoja ja suosituksia kuinka muunneltavuusominaisuudet tulisi toteuttaa. .NET-ympäristössä voidaan esimerkiksi käyttää omaa komponenttimallia [kohta 2.1.4].



Kuva 2.8. *Alusta- ja sovelluskehitys. [26]*

Alustan testaaminen saattaa olla vaikeaa, sillä alusta ei yksinään ole välttämättä ajettava. Siitä huolimatta alustaa pitäisi pyrkiä testaamaan mahdollisimman paljon uudelleenkäytettävien osien laadun maksimoinniksi. On myös mahdollista, että osa testitapauksista on uudelleenkäytettäviä sovellusta testattaessa. [25; 26, luku 4]

2.5 Organisaationäkökulma

Tuoterunkojen ja komponenttien käyttöönotto edellyttää muutakin kuin teknistä toteutusta, sillä tuoterunkojen ja komponenttien käyttöönotto on usein pitkäkestoinen ja laajamittainen prosessi. Lisäksi siirtyminen tuoterunkojen ja komponenttien käyttöön asettaa vaatimuksia organisaatiolle ja myös vaikuttaa organisaation toimintaan. Vaikutusten, vaatimusten ja ongelmien ymmärtäminen on keskeistä, koska tuoterunkojen ja komponenttien käyttöön siirtyminen voi olla hyvinkin haastavaa ja on usein myös epäonnistunut. Täysin yksikäsitteisiä toimintamalleja käyttöönoton onnistumiseen ei ole, sillä organisaatiot ovat kaikki erilaisia.

2.5.1 Vaikutukset ja organisaatiorakenteet

Eräs tavallisimmista vaikutuksista organisaation ja arkkitehtuurin välillä on niin kutsuttu Conwayn laki, jonka mukaan organisaation suunnittelema järjestelmä jäljittelee organisaation rakennetta. Tämä johtuu siitä, että komponentti annetaan tavallisesti yhden henkilön vastuulle ja vastaavasti toisiinsa liittyvät komponentit annetaan ryhmälle. Tällöin järjestelmä rakentuu organisaation rakenteen mukaan, joskin on myös mahdollista, että organisaation rakenne alkaa noudattaa arkkitehtuurin rakennetta. [27] Erityisesti tuoterunkojen kohdalla Conwayn laki tulee selkeästi esiin, sillä tuoterunkojen kehittämisaika on yksittäistä sovellusta pidempi.

Kun ohjelmistoja tuotteistetaan, organisaation tyypillinen kustannusrakenne muuttuu, koska tuotteistusasteen noustessa myös kustannukset nousevat. Kustannuksia tulee lisää muun muassa alustakehityksestä, tuki- ja hallintaprosesseista, tuotekehityksestä, markkinoinnista ja myynnistä. Toisaalta esimerkiksi asiakasrajapintaan liittyvät kustannukset saattavat pienentyä. [28, s. 22-40]

Tuoterungoille on olemassa myös suunniteltuja ja suositeltuja organisaatiorakenteita, joita voidaan jaotella sopivuuden mukaan organisaation koon ja tuotteiden suhteen. Boschin [29] mukaan nämä ovat ohjelmistokehitysosasto (development department), liiketoimintayksiköt (business unit), alustakehitysyksikkö (domain engineering unit) ja hierarkkiset alustakehitysyksiköt (hierarchical domain engineering units). Ohjelmistokehitysyksikössä (alle 30 henkilöä) kaikki työskentelevät saman tuoterungon parissa ja tällöin kommunikointi on myös usein sujuvaa. Liiketoimintayksikköihin perustuvassa mallissa (30-100 henkilöä) jokainen liiketoimintayksikkö kehittää ja ylläpitää omaa sovellustaan. Alustakehitysyksikköön pohjautuvassa mallissa (yli 100 henkilöä) tuotekehitys on irrotettu sovelluskehityksestä, jotta saavutettaisiin selkeämpi vastuunjako alustan ja sovellusten välille. Hierarkkisiin alustakehitysyksiköihin perustuvalla rakenteella pyritään puolestaan hallitsemaan paremmin erikoistuneita tuoterunkoja, jotka saattavat poiketa paljonkin perusalustasta. [29]

2.5.2 Riskejä ja ongelmakohtia

Tuoterunkojen haittoja on käsitelty jo kohdassa 2.4.1, mutta keskitytään vielä erityisesti organisaation kannalta keskeisiin riskeihin ja ongelmakohtiin, sillä siirtyminen projekti- tai palveluliiketoiminnasta kohti tuoteliiketoimintaa ja tuoterunkoja ei useinkaan ole triviaalia vaan siirtymisprosessi sisältää useita haasteita.

Taloudellista näkökulmasta tuoterungoilta saatetaan odottaa vallankumouksellista tuottoa, sillä tuoterunkojen avulla voidaan päästä uusille markkinasegmenteille pienillä kustannuksilla ja vähillä riskeillä [30]. Tässä tulee kuitenkin huomioida, että tuoterunkoihin ja komponentteihin siirtyminen on pitkä prosessi, ja tuottojen saaminen saattaa kestää pitkäänkin. Eriksson [31] esittääkin, että tuoterungosta tulee rakentaa vähintään 2-3 tuotetta ennen kuin tuotekehityksen kulut saadaan katettua. Tämän takia tuoterungot ja komponentit eivät sovellu erityisen hyvin kvartaalitalouteen.

Lisäksi tuoterungot edellyttävät organisaation sisäistä kommunikointia, vastuunjakoa, halua ja pitkäjänteistä työtä (etenkin arkkitehtuurin suhteen) sekä kokonaisvaltaista ymmärrystä tuoterungoista. Kommunikoinnin ja tiedonkulun tulee olla sujuvaa eri kehitysyksiköiden välillä, mutta myös tuotekehityksen sekä myynnin ja markkinoinnin välillä. Mitä suurempi organisaatio on kyseessä, sitä enemmän kommunikointiin luonnollisesti tulee panostaa. Tuotteistaminen edellyttää yhteistyötä – niin kutsutuilla sankariteoilla ei rakenneta pitkäjänteistä tuoterunkoa, joten henkilöstön tulee ymmärtää oma rooli ja vastuualue. Lisäksi henkilöstön mahdollinen muutosvastarinta tulee voittaa. Tuotteiden suhteen sovelluskehittäjiltä vaaditaan myös kokonaisvaltaista osaamista, sillä toisin kuin projektiliiketoiminnassa, tuotteiden kohdalla asiakkaan kanssa ei voida

käydä pitkiä palavereita suunniteltavan tuotteen määrittelemiseksi. [4, s. 173-175; 29; 31].

Lisäksi tekniset piirteet asettavat vaatimuksia organisaatioille, sillä tuotteistaminen edellyttää oletettavasti myös uusien asioiden opettelua. Käytettävät komponenttitekniikat tulee hallita hyvin ja pitkäikäiset alustat vaativat huolellista suunnittelua. Henkilöstön tulee hallita myös tuotekehityksessä käytettävät prosessit. Lisäksi uudelleenkäytön onnistuminen voidaan todeta vasta kehitystyön jälkeen ja toisaalta ensimmäinen tuoterunkoon perustuva tuote ei välttämättä ole menestys. [13, s. 33-37; 31]

3 NYKYISEN JÄRJESTELMÄN ARKKITEHTUURI

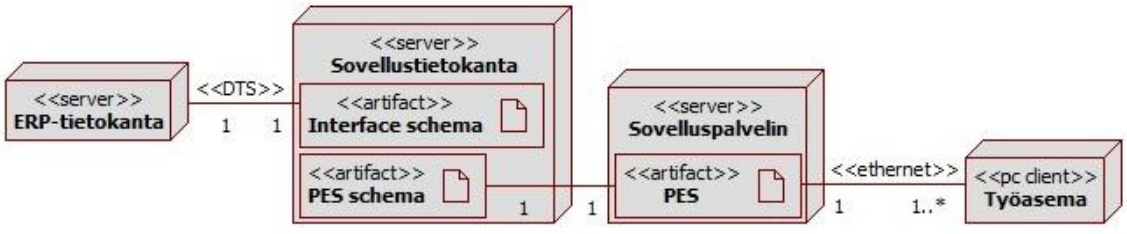
Tutkittava järjestelmä on yksi SW-Developmentin rakentamista Planning Efficiency System eli PES-tuoteperheen järjestelmistä. Järjestelmä on kehitetty .NET-ympäristössä C#-ohjelmointikieltä käyttäen. PES-sovellus on asiakaskohtaisesti räätälöitävä, Windows-ympäristössä toimiva tuotannonsuunnittelutyökalu, jonka tavoitteena on parantaa asiakasorganisaation tuotannon ja toiminnan tehokkuutta. Tuoteperheen sovellusten keskeisimpiä ominaisuuksia ovat tuotannon suunnittelu, erilaiset suunnitteluskenaariot (tai suunnitelmavaihtoehdot), raportointi, optimointi, simulointi ja integraatio organisaation toiminnanohjausjärjestelmään (ERP). Kaikki järjestelmät eivät sisällä kaikkia ominaisuuksia ja toisaalta ominaisuudet ovat asiakaskohtaisesti räätälöityjä, jolloin niiden toiminta saattaa järjestelmien välillä vaihdella paljonkin. Järjestelmän käyttöliittymäkuva on liitteessä 1.

3.1 Arkkitehtuurikuvaus

Arkkitehtuurikuvauksella pyritään selvittämään nykyisen järjestelmän rakennetta, sillä tutkittavalle järjestelmälle ei ole tehty arkkitehtuurisuunnitelmaa tai suunnitteludokumenttia. Tällöin joudutaan tekemään takaisinmallinnusta, jotta saataisiin käsitys järjestelmän toiminnasta. Takaisinmallinnusta tehdään tutkimalla järjestelmän lähdekoodia, rakentamalla erilaisia kaavioita (ohjelmallisesti ja käsin), analysoimalla järjestelmän metriikoita sekä haastattelemalla järjestelmän rakennukseen osallistuneita henkilöitä. Tässä arkkitehtuurikuvauksessa painopiste on järjestelmän sisäisessä rakenteessa, järjestelmän osien vuorovaikutuksessa sekä järjestelmässä käytettävien suunnitteluratkaisujen tutkimisessa.

3.1.1 Fyysinen rakenne

Sijoittelukaavio kuvaa kuinka järjestelmä sijoittuu käytettäville laitealustoille. Järjestelmän sijoittelukaavio on kuvattu kuvassa 3.1. Sijoittelukaaviosta havaitaan, että järjes-

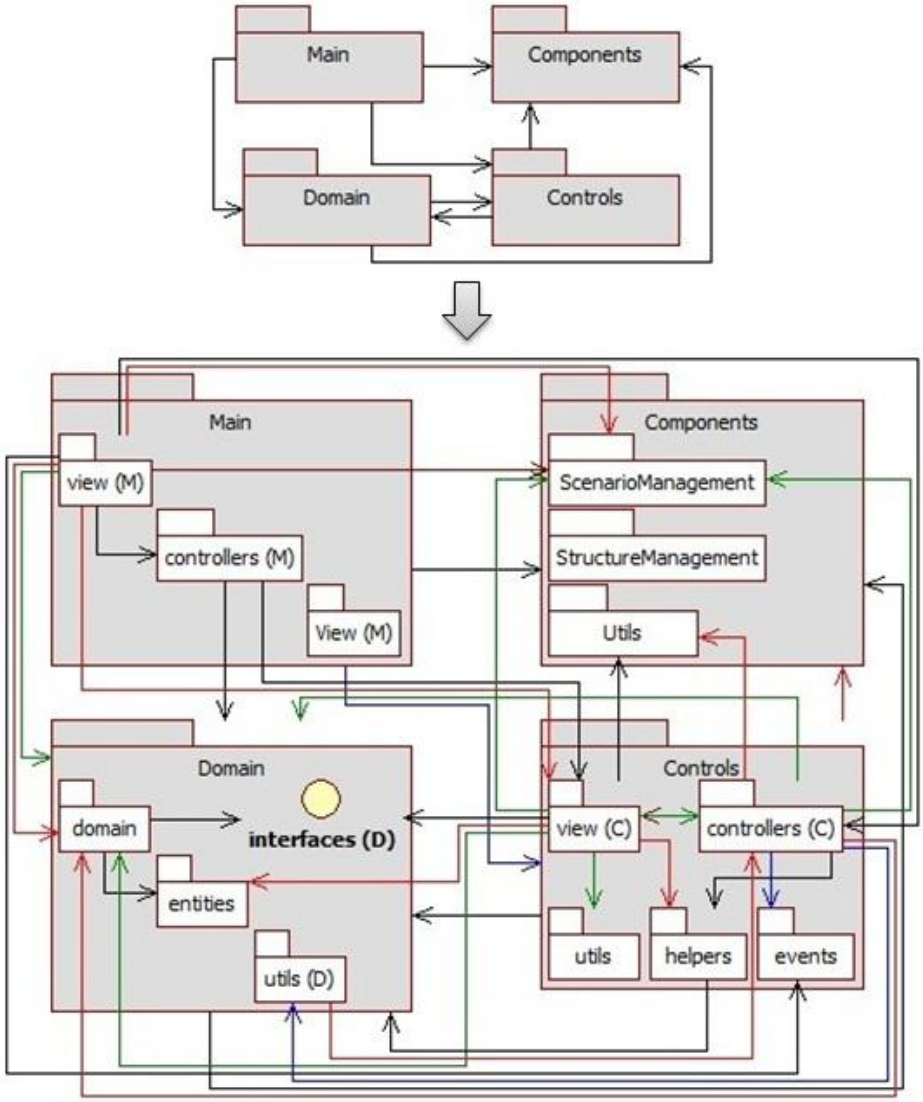


Kuva 3.1. *Tutkittavan järjestelmän sijoittelukaavio.*

telmä toimii omalla sovelluspalvelimellaan, jota voidaan käyttää tietoverkon yli useammalla työasemalla. Sovelluspalvelimella on yhteys sovellustietokantaan, jossa sovellus käyttää omaa tietokantarakennetta. Sovellustietokanta on edelleen yhdistetty organisaation ERP-tietokantaan, jolloin erillisellä rajapintarakenteella voidaan hyödyntää ERP-järjestelmän käyttämä tieto myös PES:ssä. Tällöin puhutaan ERP-integraatiosta, jolloin esimerkiksi tuotantotiedot voidaan hakea ERP-järjestelmästä, käsitellä PES-sovelluksessa ja tallentaa takaisin tietokantaan.

3.1.2 Looginen rakenne

Pakkauskaavio kuvaa järjestelmän sisältämien pakkausten välisiä suhteita. Järjestelmän pakkauskaavio on esitetty kuvassa 3.2. Kuvassa 3.2 käytettävät eri värit assosisaatioiden



Kuva 3.2. *Tutkittavan järjestelmän pakkauskaavio kahdella eri abstraktiotasolla.*

merkitsemiseen on tehty vain luettavuuden parantamiseksi (värit eivät siis kuvaa suhteita millään tavalla). Ylemmällä abstraktiotasolla on kuvattu järjestelmän sisältämät projektit (projekti on .NET-ympäristössä käytettävä nimitys ylimmän tason pakkaukselle).

Alemmalle abstraktiotasolle on lisätty projektien sisältämät kansiot. Esimerkiksi Main-projektista lähtee nuoli Components-projektiin, mikä tarkoittaa, että Main-projektissa on luokka, joka käyttää jotain Components-projektin luokkaa. Jokainen projekti ja kansio luovat oman nimiavaruuden eli esimerkiksi Controls-projekti luo nimiavaruuden SWDeve.SOP.Controls, jossa SWDeve.SOP on koko järjestelmän nimi. Vastaavasti Controls-projektissa oleva view-kansio luo nimiavaruuden SWDeve.SOP.Controls.view. Nimeämiskäytäntö toimii vastaavalla tavalla aina luokkatasolle asti. Pakkauskaavioihin on myös merkitty rajapintatiedostot (interfaces), jotka ovat käytännössä C#-ohjelmointikielen tapa toteuttaa erillisiä rajapintoja. Nämä rajapinnat poikkeavat luokista muun muassa siten, että rajapintoihin ei saa määritellä muuttujia tai funktioita. Järjestelmän eri osien keskinäiset assosisaatiot on luettavissa kuvista 3.1 ja 3.5. Järjestelmän toiminnallisuus on jaettu kuvien 3.1 ja 3.5 kuvaamiin osiin. Toiminnallisia ominaisuuksia on kuvattu lyhyesti taulukossa 3.1.

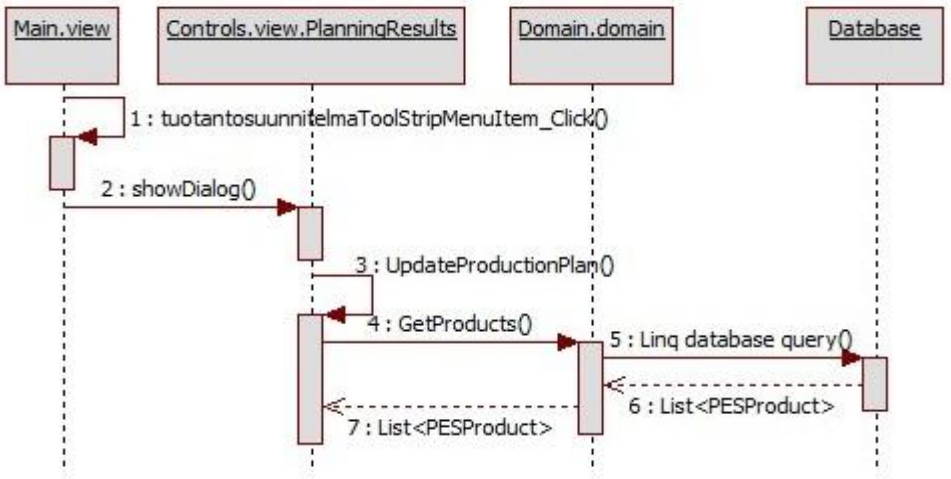
Taulukko 3.1. *Järjestelmän osien toiminnallisuudet.*

Main	Ohjelman main-lohko, käynnistää sovelluksen
Main.view	Sovelluksen päänäkymä
Main.controllers	Päänäkymän skenaariotyökalun kontrollit
Main.View	Päänäkymän sivuikkuna
Domain.utils	Erilliset tietorakenteet
Domain.interfaces	Rajapintatiedostot varastoille, tuotteille ja toimipaikoille
Domain.entities	Konkreettisia asioita kuvaavat tietokantaentiteetit
Domain.entities.customEntities	Lisäentiteetit
Domain.domain	Tietokantakäsittelijä, suorittaa tietokannan transaktiot
Domain	Määrittelee tietuetyypit tietokannan datalle
Controls.view	Lukittujen skenaarioiden näkymä
Controls.view.WizardControls	Uuden skenaarion lisäämiseen liittyvät näkymät
Controls.view.WizardControls.subcontrols	Uuden skenaarion lisäämiseen liittyvät alinäkymät
Controls.view.ScenarioDataControl	Varasto- ja resurssinäkymät
Controls.view.PlanningResults	Tuotantosuunnitelman näkymä
Controls.view.PlanningControls	Gantt-suunnittelutyökalun näkymä
Controls.view.MasterDataControls	Pääsuunnitelman ja suunnitteluvaihtoehtojen näkymä
Controls.utils	Kalenterinäkymän kontrolloinnin lisämääritykset
Controls.Helpers	Apufunktioita
Controls.events	Suunnitteluvaihtoehdon poistaminen
Controls.controllers	Perustoimintalogiikka
Controls.controllers.ScenarioDataControllers	Varasto- ja resurssihallinan logiikka
Controls.controllers.PlanningControls	Tuotanto- ja Gannt-suunnitelman logiikka
Controls.controllers.MasterDataControllers	Pääsuunnitelman ja suunnitteluvaihtoehtojen logiikka
Controls.controllers.tabPageControllers	Ennusteisiin ja kapasiteetteihin liittyvä logiikka
components/Utils	Aikajaksojen käsittelyyn liittyviä apufunktioita
components/StructureManagement	<Tyhjä pakkaus>
components/ScenarioManagement	Skenaarionhallinnan logiikka
components/ScenarioManagement.utils	Skenaariohallinnan apufunktioita
components/ScenarioManagement.interfaces	Skenaariohallinnan rajapintatiedosto

Järjestelmän on suunniteltu noudattavan Model-View-Controller eli MVC-arkkitehtuurityyliä [32]. Tutkittavassa järjestelmässä kyseinen MVC-malli ei kuitenkaan toteudu täysin. Järjestelmässä Model-osaa vastaa Domain-pakkaus, View-osa (järjestelmässä Controls.view) on sisällytetty Controls-pakkaukseen ja myös Controller-osa on sisällytetty Controls-pakkaukseen (järjestelmässä Controls.controllers) [kuva 3.3]. Main-pakkaukseen on myös merkitty view-, View-, ja controllers-kansiot, jotka osaltaan toteuttavat MVC-mallia käyttöliittymän päänäkymän osalta. Järjestelmässä ei käytetä Observer-mallia (tai muuta vastaavaa) ja toisaalta myös kutsusuhteita rikotaan verrattuna puhtaaseen MVC-malliin. Tutkittavan järjestelmän rakenne on oikeastaan lähempänä kerrosarkkitehtuuria, jossa tehdään ohituksia (kuten kuvasta 3.2 voi havaita).

3.1.3 Vuorovaikutus

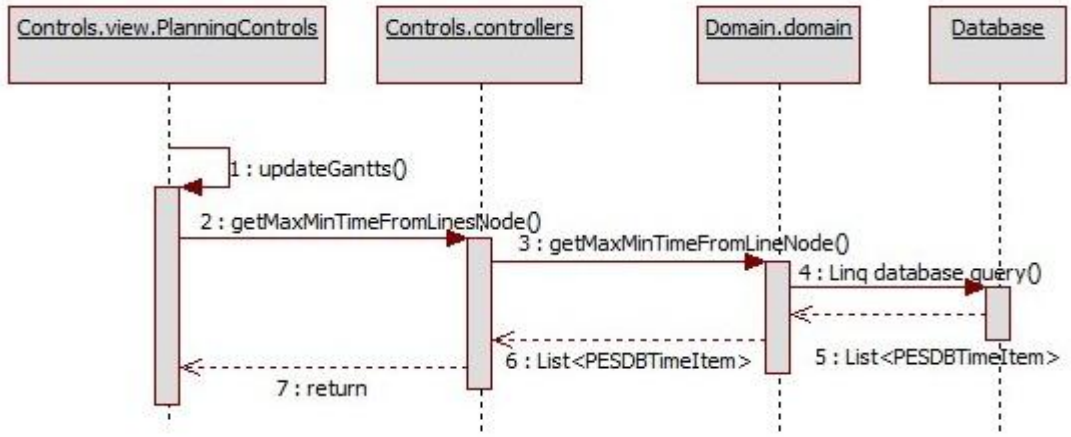
Sekvenssikaavion tarkoitus on kuvata olioiden välistä vuorovaikutusta. Pystysuorat viivat kuvaavat olion elinikää ja vaakasuorat viivat puolestaan olioiden välistä kommunikointia siten, että kiinteät viivat ovat kutsuja (call) ja katkoviivat kuvaavat paluuta kutsujaan (return). Sekvenssikaavioissa on tavallisesti kuvattuna olioita, mutta kuvissa 3.3 ja 3.4 oliot on abstrahoitu pois (laatikoissa näkyy vain käytettävä nimiavaruus), koska tässä tapauksessa sekvenssikaavioilla halutaan selvittää järjestelmän eri osien vuorovaikutusta eikä siten ole tarvetta mennä luokka- tai oliotasolle. Tällöin voidaan ajatella, että esimerkiksi kuvan 3.3 mukaan Main.view-nimiavaruus sisältää jonkin luokan, jolla on funktio tuotantosuunnitelmaToolStripMenuItem_Click(). Myös vuorovaikutuksen kannalta tarpeettomia funktioita on abstrahoitu pois selkeyden parantamiseksi. Tällaisia funktioita ovat muun muassa luokan omat tarkistusfunktiot. Tietokanta (kaavioissa Database) ei myöskään ole oma olio tai nimiavaruus vaan yksinkertaisesti kuvaa järjestelmän käyttämää tietokantaa, josta tehdään kyselyitä LINQ:a apuna käyttäen.



Kuva 3.3. *Tuotantosuunnitelman päivityksen sekvenssikaavio.*

Kuvassa 3.3 esitetään tuotantosuunnitelman päivitys. Ensimmäinen kutsu on Main.view-tasolta (1), josta avataan uusi dialogi, jossa näytetään tuotantosuunnitelma (2). Tässä dialogissa on suunnitelman päivityskuvake, jota painamalla suunnitelman voi päivittää tietokannasta (3). Dialogista kutsu etenee Domain.domain-tasolle (4), jossa

suoritetaan tietokantakysely (5). Tiedot tuotteista palautetaan ensin Domain-tasolle (6) ja siitä edelleen Controls-tasolle (7), jossa näkymä päivitetään. Huomattavaa on, että tässä hypätään näkymätasolta suoraan datatasolle ohittaen toimintalogiikka.



Kuva 3.4. *Gantt-näkymän päivityksen sekvenssikaavio.*

Kuvassa 3.4 esitetään Gannt-näkymän päivitys. Ensimmäinen kutsu on Controls.view.PlanningControls-tasolta, jossa on Gantt-näkymän päivityksen kuvake (1). Gantt-näkymä on toteutettu erillisenä ja integroitu päänäkymään vaikka sijaitseekin eri projektissa. Tältä näkymätasolta kutsu etenee Controls.controllers-tasolle (2) ja siitä edelleen Domain.domain-tasolle (3). Domain-tasolla suoritetaan tietokantakysely kuten kuvan 3.3 kaaviossakin (4). Paluuarvo välitetään Controls-tasolle (5, 6), jossa päivityksen logiikka suoritetaan. Lopuksi näkymä päivittyy käyttäjälle (7). Erona kuvan 3.3 kaavioon, tässä kuljetaan kerrosarkkitehtuurin kannalta jokaisen tason läpi ilman ohituksia.

3.1.4 Metriikat

Metriikat mittaavat järjestelmän ominaisuuksia ja antavat siten käsitystä järjestelmän rakenteesta ja ominaisuuksista. Tutkittavan järjestelmän metriikat on koottu ohjelmallisesti Visual Studio –kehitysympäristön työkaluja käyttäen. Järjestelmän metriikat on esitetty taulukossa 3.2. Tutkittaviksi metriikoiksi on valittu järjestelmän rakennetta kuvaavia ominaisuuksia. Syklomaattinen kompleksisuus kuvaa järjestelmän monimutkaisuutta siten, että jokainen mahdollinen itsenäinen suorituspolku kasvattaa arvoa yhdellä. Jokainen haaraumakohta, esimerkiksi if-else–lause, kasvattaa siis arvoa. Perintäsyvyys tarkoittaa, kuinka monta kertaa luokka on periytetty jostain toisesta luokasta tai rajapinnasta. Luokkasidonnaisuus puolestaan tarkoittaa, kuinka monta kertaa luokasta viitataan johonkin toiseen luokkaan. Taulukossa 3.2 käsitellään kuitenkin kokonaisia järjestelmän osia eli taulukko kuvaa, kuinka monta kertaa jonkin järjestelmän osan luokista viitataan johonkin toisiin luokkiin. Rivimäärä tarkoittaa kuinka monta koodiriviä järjestelmän osa sisältää. Koodirivejä laskettaessa kommentit, viittaukset (kuten tiedoston alussa olevat using-määreet) ja koodilohkoja kuvaavat merkit (käytännössä rivit, jotka sisältävät yhden aaltosulun) on jätetty pois. Kuitenkin muun muassa funktioiden esittely on laskettu mukaan koodirivien kokonaismäärään.

Taulukko 3.2. *Tutkittavan järjestelmän metriikat.*

Järjestelmän osa	Syklomaattinen kompleksisuus	Perintä-syvyys	Luokkasi-donnaisuus	Rivimäärä
Main	68	7	120	1031
Main.view	5	1	1	5
Main.controllers	1	1	3	3
Main.View	12	7	24	260
Main yhteensä	**86**	**7**	**122**	**1299**
Domain.utils	30	3	21	49
Domain.interfaces	23	0	0	0
Domain.entities.customEntities	18	1	0	18
Domain.entities	13	1	2	15
Domain.domain	251	1	49	1581
Domain	1685	2	72	3300
Domain yhteensä	**2020**	**3**	**105**	**4963**
Controls.view.WizardControls.subcontrols	51	7	54	208
Controls.view.WizardControls	442	8	186	2543
Controls.view.ScenarioDataControl	19	7	48	271
Controls.view.PlanningResults	31	7	56	190
Controls.view.PlanningControls	423	7	156	2924
Controls.view.MasterDataControls	75	7	81	1285
Controls.view	6	7	25	129
Controls.utils	30	5	18	52
Controls.Helpers	68	1	19	79
Controls.events	7	2	4	3
Controls.controllers.tabPageControllers	184	1	76	1032
Controls.controllers.ScenarioDataControllers	29	1	12	49
Controls.controllers.PlanningControls	150	1	58	690
Controls.controllers.MasterDataControllers	94	1	46	215
Controls.controllers	89	1	85	763
Controls yhteensä	**1798**	**8**	**343**	**10433**
components/Utils	34	1	16	77
components/StructureManagement	0	0	0	0
components/ScenarioManagement	244	7	96	634
components/ScenarioManagement.utils	6	2	2	6
components/ScenarioManagement.interfaces	12	1	4	0

Syklomaattista kompleksisuutta tarkasteltaessa huomataan, että Domain-osalla on ylivoimaisesti suurin arvo (1685, seuraavaksi suurin on 442). Tämä johtuu suurilta osin siitä, että Domain-osassa määritellään tietuetyypit ja suurin osa koodista on automaattisesti generoitua .NET-työkaluja käyttäen – suuri arvo ei siis suoraan tarkoita, että kyseinen osa olisi erityisen monimutkainen. Tämän jälkeen suurimpia arvoja on Controls.view.WizardControls- (442), Controls.view.PlanningControls- (423), Domain.domain- (251), ja Components/ScenarioManagement-osilla (244). Muilta osin syklomaattinen kompleksisuus on jakautunut melko tasaisesti välille 0–184.

Perintäsyvyys on suurimmillaan 7. Eri osien view-osat omaavat suurimmat perintäsyvyydet, mikä johtuu siitä, että näkymiä on periytetty .NET-ympäristön valmiista luokista, kuten lomakkeista (form). Logiikkaa tai datankäsittelyä sisältäviä luokkia ei käytännössä ole periytetty.

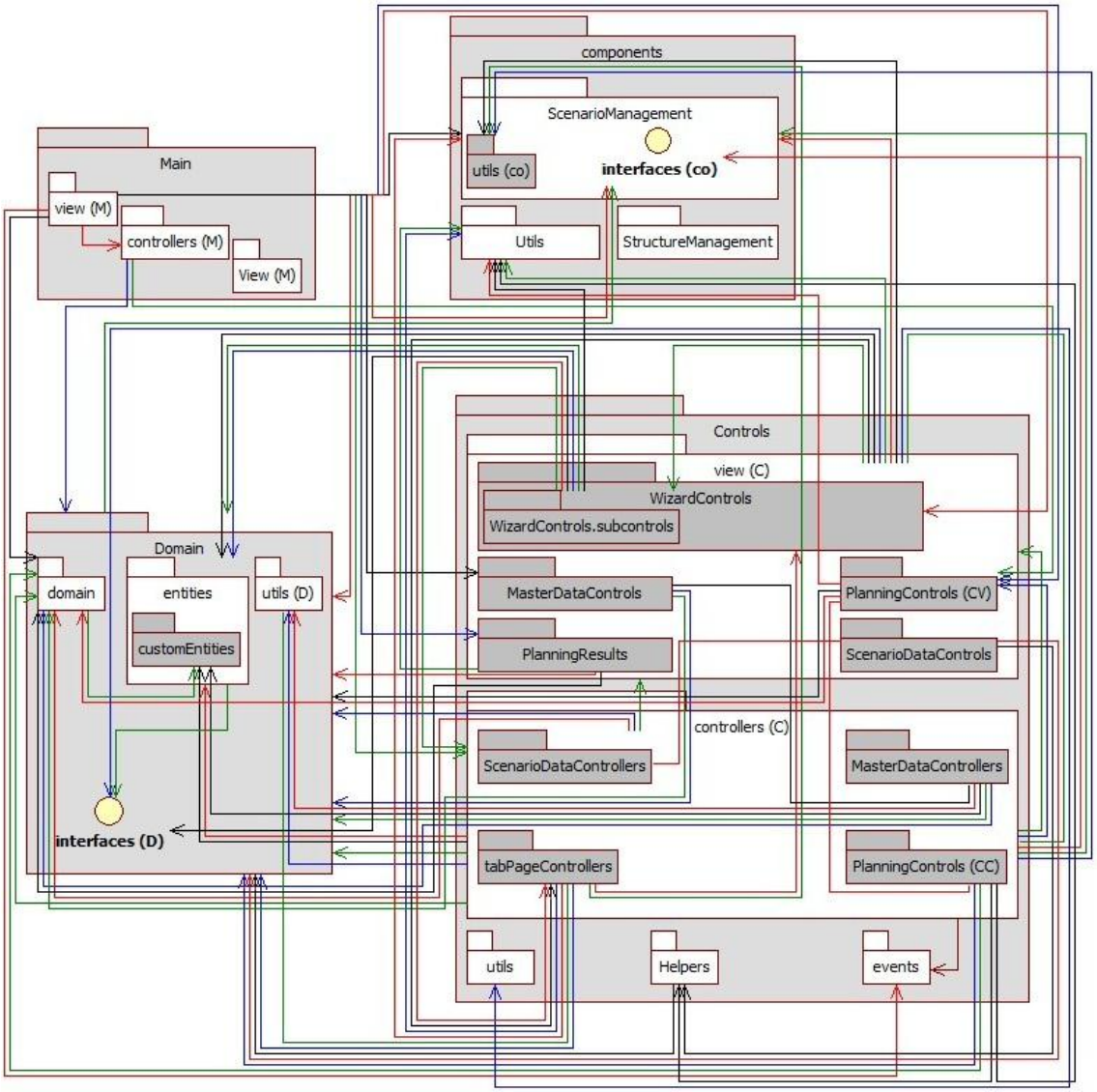
Luokkasidonnaisuus ja rivimäärät korreloivat selkeästi syklomaattisen kompleksisuuden kanssa. Tämä viittaa siihen, että järjestelmä rakentuu hyvin pitkälti tiettyjen järjestelmän osien varaan, mikä usein viittaa epätasapainoiseen järjestelmään. Erityisesti osilla Main, Controls.view.WizardControls ja Controls.view.PlanningControls luokkasidonnaisuus on hyvin korkea: 120–186. Muilta osin luokkasidonnaisuus vaihtelee välillä 0–96 keskiarvon ollessa 32. Rivimäärien suhteelliset osuudet koko järjestelmästä (17 412 riviä) on jakautunut seuraavasti: Main 7 %, Domain 29 %, Controls 60 % ja components 4 %. Rivimäärien osalta huomiota herättävät samaiset osat kuin syklomaattisen kompleksisuuden osalla. Main-nimiavaruus kattaa 79 % koko Main-paketin rivimäärästä (vaikka nimet ovat samat, niin ne tarkoittavat eri osaa järjestelmästä, sillä Main-paketti sisältää luokkia sekä kansioita ja Main-nimiavaruus sisältää suoraan Main-paketissa olevat luokat poissulkien kansioiden sisältämät luokat). Vastaavasti Controls.view.WizardControls- ja Controls.view.PlanningControls-osat kattavat 52 % koko Controls-osan riivimäärästä.

3.2 Ongelmakohtia

Arkkitehtuurikuvauksesta voidaan havaita tiettyjä arkkitehtuuriin liittyviä ongelmakohtia, joita on hyvä tunnistaa, jotta komponenttipohjaisen arkkitehtuurin rakentaminen onnistuu paremmin. Etenkin, jos nykyistä järjestelmää käytetään pohjana tuleville järjestelmille, niin on tärkeää tunnistaa sen mahdollisia heikkouksia. Ensimmäisenä ongelmakohtana voidaan pitää dokumentoinnin puutetta, sillä jos arkkitehtuuria ei ole dokumentoitu, niin arkkitehtuuria ei oikeastaan ole olemassa. Tämän takia on riskialtista käyttää nykyistä järjestelmää pohjana, sillä arkkitehtuuria voidaan pitää tietynlaisena järjestelmän perustuslakina. Lisäongelmia tuottaa se, että vaikka arkkitehtuuri olisikin dokumentoitu, niin ylläpidon ja kehityksen myötä arkkitehtuuri yleensä rapautuu ja tilanne on entistä vaikeampi, mikäli arkkitehtuuria ei ole alun perinkään määritelty. [4, s. 20] Takaisinmallinnuksella voidaan kuvata järjestelmää, mutta takaisinmallinnus ei tuo kattavasti esiin suunnitteluratkaisuja ja niiden perusteita etenkään kun suunnitteludokumenttia ei ole tehty.

Osaltaan dokumentointiin liittyvä ongelma on selkeän arkkitehtuurityylin puuttuminen. Järjestelmän on tarkoitus käyttää MVC-mallia, mutta järjestelmän kuvaus ei ole yhtäpitävä kyseisen mallin kanssa vaan muistuttaa pikemminkin kolmikerrosarkkitehtuuria. Tämän takia etenkin ulkopuolisen henkilön on vaikeampaa saada käsitystä järjestelmän perusrakenteesta ja –ideasta. Tällöin myös tulevat kehitysratkaisut ja muutokset saattavat ohjautua väärään suuntaan ja olla ristiriidassa järjestelmän muiden ratkaisujen kanssa. Muun muassa kuvat 3.2 ja 3.3 herättävät kysymyksen, milloin ohituksia kerrosten välillä on suotavaa tehdä.

Kuvat 3.2 ja 3.5 sekä taulukko 3.2 tuovat esille järjestelmän osien melko vahvan sidonnaisuuden ja epätasapainon. Kuvassa 3.5 on tehty vielä tarkempi näkymä kuvan 3.2 pakkauskaaviosta. Kuvassa 3.5 näkyy myös alimman tason pakkaukset ja jos toteutettaisiin vielä alemman tason kuvaus, niin kyseisessä kuvauksessa näkyisi myös luokat



Kuva 3.5. *Tutkittavan järjestelmän pakkauskaavio alimmalla abstraktiotasolla.*

Kuva 3.5 tuo esiin järjestelmän eri osien sidonnaisuuden ja kutsusuhteet vielä tarkemmalla tasolla. Kuvauksessa on noudatettu samoja periaatteita kuin kuvassa 3.2. Yleinen periaate ohjelmistoissa on ”high cohesion, loose coupling” eli olisi suotavaa rakentaa yhtenäisiä osia, jotka eivät ole sidoksissa moniin muihin osiin. Vastaavasti erityisen suuret järjestelmän osat viittaavat God object–antisuunnittelumalliin, joka tarkoittaa, että jokin objekti sisältää liian suuren osan järjestelmän toiminnallisuudesta tai tuntee järjestelmän rakennetta tarpeettoman paljon.

Osaltaan vahvaan sidonnaisuuteen ja epätasapainoon liittyvä ongelma on järjestelmän modulaarisuuden ja selkeän vastuunjaon puute. Järjestelmä on jaettu pienempiin osiin, mutta mikäli tietyt osat kattavat kohtuuttoman suuren osan toiminnallisuudesta, niin herää kysymys, onko jako tehty oikein tai voitaisiinko suurimpia kokonaisuuksia

jakaa vielä pienempiin osiin hajota ja hallitse –periaatteen mukaan. Järjestelmästä on myös vaikea tunnistaa varsinaisia komponentteja, jotka vastaisivat yksinomaan tietyn ominaisuuden toiminnallisuudesta.

Järjestelmän nimeämiskäytäntö on myös osittain sekava, sillä järjestelmässä on useita view-, utils- ja controls-nimisiä osia sekä lisäksi controls- ja controllers-nimet saattavat sekoittua. Lisäksi järjestelmässä on viitattu tarpeettomasti eri osiin mikä kasvattaa taulukossa 3.2 ja kuvissa 3.2 ja 3.5 ilmenevien viittausten määrää. Tarpeettomalla viittauksella tarkoitetaan viittausta johonkin toiseen osaan vaikka todellisuudessa kyseistä järjestelmän osaa ei käytetä ohjelmakoodissa mihinkään. Tämä rapauttaa järjestelmää ja tekee rakenteesta vaikeammin ymmärrettävän. Järjestelmässä on myös StructureManagement-nimiavaruus, joka ei kuitenkaan sisällä mitään. Tämä herättää kysymyksen, että onko jotain poistettu, onko nimiavaruus tarpeeton vai vain keskeneräinen. Myös tällä on järjestelmää rapauttava vaikutus.

4 VAATIMUKSET

Jotta järjestelmän arkkitehtuuri voidaan suunnitella, tulee ensin käsitellä järjestelmälle asetetut vaatimukset. Vaatimuksissa tulee ottaa huomioon organisaation asettamat yleiset vaatimukset, toiminnalliset vaatimukset ja tuoterungon asettamat vaatimukset sekä tunnistaa tärkeimmät laatuominaisuudet. Asetettujen vaatimusten täyttäminen on luonnollisesti suunniteltavan arkkitehtuurin lähtökohta.

4.1 Yleiset vaatimukset

Tavoitteena on suunnitella Windows-ympäristössä toimiva, .NET-kirjastoa käyttävä, tuoterunkoajattelua soveltava arkkitehtuuri, joka on muunneltava ja mahdollistaa komponenttien uudelleenkäytön. Lisäksi arkkitehtuuri ei saa olla liian erilainen nykyisen järjestelmän arkkitehtuurin verrattuna ja arkkitehtuurin tulee noudattaa yleisiä suunnittelusääntöjä. Näistä tavoitteista voidaan johtaa arkkitehtuurille asetettuja vaatimuksia.

Tavoite Windows-ympäristön ja .NET-kirjaston suhteen johtuu siitä, että SW-Development on valinnut Microsoftin yhteistyökumppanikseen ohjelmistokehityksen osalta. Lisäksi suurin osa SW-Developmentin asiakkaista käyttää Windows-käyttöjärjestelmiä. Näistä tavoitteista muodostuva vaatimus on siten Visual Studio –ohjelmiston käyttö ohjelmistokehityksessä. Muita järjestelmiä voidaan käyttää myös apuna (esimerkiksi erilliset simulointiohjelmat), mutta tällöin tulee huomioida myös niiden integroitavuus suunniteltavaan järjestelmään.

Tuoterunkoajattelun soveltaminen perustuu siihen, että järjestelmillä on havaittavissa yhteisiä piirteitä, jolloin on edullisempaa käyttää yhteistä tuoterunkoa. Tästä johdettava vaatimus on, että arkkitehtuurin tulee tukea tuoterunkoajattelua eikä siten saa olla sidottu yksittäisiin järjestelmiin. Organisaatiossa tuoterungon lähtökohta on valita yksi tuote mallituotteeksi, jota pyritään kehittämään täysimittaiseksi tuoterungoksi [kohta 2.4.2].

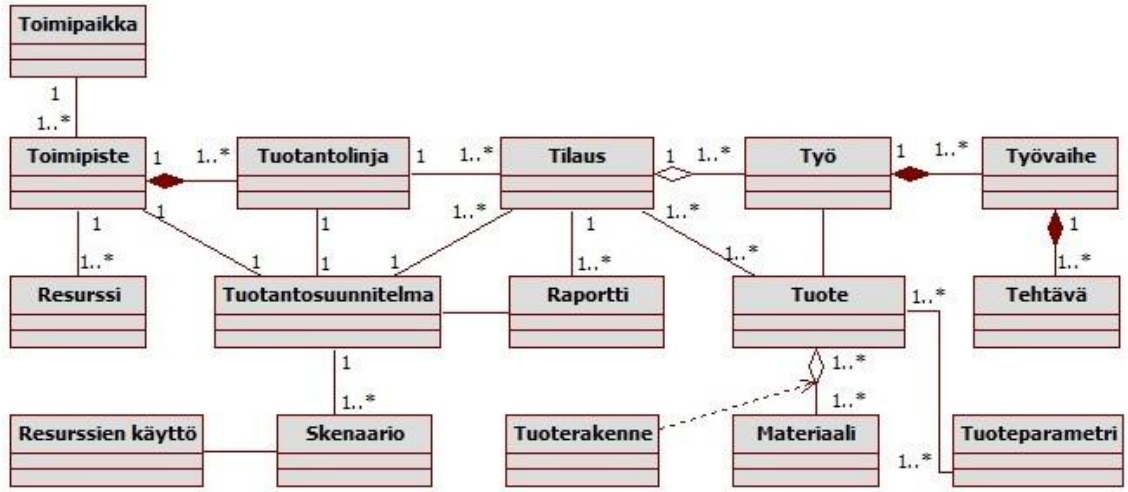
Muunneltavuus ja komponenttien käyttö on suunniteltavan järjestelmän keskeisimpiä laatuominaisuuksia. Vaikka järjestelmät sisältävät yhteisiä ominaisuuksia, niin järjestelmille tehdään myös aina asiakaskohtaista konfigurointia. Tällöin arkkitehtuurin tärkein laatuvaatimus on juuri muunneltavuus ja lisäksi komponenteista luotavat tuotevariaatiot tulee olla mahdollista toteuttaa arkkitehtuurin avulla.

Arkkitehtuuri ei saa olla liian erilainen nykyisen järjestelmän arkkitehtuurin verrattuna, sillä organisaatioon on muodostunut vakiintunut tapa toteuttaa järjestelmiä. Jos arkkitehtuuri poikkeaa liikaa nykyisen järjestelmän arkkitehtuurista, niin uusi arkkitehtuurisuunnitelma aiheuttaa luultavasti tarpeettoman korkean oppimiskynnyksen ja toi-

saalta myös muutosvastarintaa. Tästä voidaan johtaa vaatimus, että arkkitehtuurityylin tulee olla kerrosarkkitehtuuri.

4.2 Käsitemalli

Käsitemalli on kuvaus järjestelmän toiminta-alueen käsitteistä, niiden ominaisuuksista, suhteista ja rooleista. Käsitemallin avulla luodaan myös yhteinen sanasto ja pyritään varmistumaan, että kaikki osapuolet ymmärtävät sovellusalueen samalla tavalla. [4, s. 165] Suunniteltavan järjestelmän käsitemalli on esitetty kuvassa 4.1.



Kuva 4.1. *Suunniteltavan järjestelmän käsitemalli.*

Kuvan 4.1 käsitemallissa tulee huomata, että se ei kuvaa yksittäistä järjestelmä – kuten jotain vanhempaa järjestelmää – sillä yksittäisten järjestelmien käsitemallit eivät ole täysin samanlaiset. Tämän takia onkin hyvä luoda käsitemalli tuoterunkoa ajatellen. Kuvaa 4.1 voidaan täsmentää seuraavilla kuvauksilla:

- Toimipiste kuuluu Toimipaikkaan ja Toimipaikassa voi olla useita Toimipisteitä.
- Resurssi on Toimipistekohtainen.
- Toimipisteellä voi olla useita Tuotantolinjoja.
- Tuotantosuunnitelma rakentuu Tilauksista ja Toimipisteellä sekä Tuotantolinjalla voi olla Tuotantosuunnitelma.
- Raportti voidaan luoda Tilauksesta tai Tuotantosuunnitelmasta (tai niiden toteutumisesta).
- Työ muodostuu Työvaiheista, joka muodostuu edelleen Tehtävistä. Työ liittyy yhteen Tilaukseen.
- Tilaukseen voi liittyä useita Tuotteita ja Tuotteeseen voi koostua useista Materiaaleista, lisäksi Tuotteeseen kuuluu Tuoterakenne ja Tuoteparametrit.
- Tuoterakenteella (BOM, bill of materials) tarkoitetaan Materiaalien (joskus myös tuoteosien) hinta- ja määräkomponentteja.
- Tuotantosuunnitelmasta voidaan tehdä useita Skenaarioita.
- Skenaario on käytännössä Resurssien käyttöä, eri Skenaarioissa Resurssien käyttö on vain erilaista.

On myös mahdollista, että käsitemallia tarkennetaan tai muutetaan yksittäisen järjestelmän kohdalla.

4.3 Toiminnalliset vaatimukset

Toiminnallisten vaatimusten määrittelyssä käytetään vanhojen järjestelmien toiminnallisuutta. Yhteiset ominaisuudet ovat lähtökohta mahdollisen tuoterungon rakentamiselle ja komponenttien tunnistamisella voidaan pohtia, mitä komponentteja olisi kannattavaa toteuttaa ja miten komponentit toteutettaisiin. Muita tuoteperheen järjestelmiä ei kuitenkaan tutkita yhtä tarkasti kuin luvussa 3 käsiteltyä järjestelmää, sillä muut järjestelmät ovat vanhempia. Lisäksi analyysissa otetaan kantaa sen suhteen, että millaisia järjestelmiä tulevaisuudessa olisi tarkoitus toteuttaa.

Taulukossa 4.1 on esitetty järjestelmien tavallisimpia ominaisuuksia, ominaisuuksien kuvaukset ja lukumäärä kuinka monessa järjestelmässä kyseiset ominaisuudet oli tunnistettavissa. Analysoinnissa on käsitelty kolmea eri järjestelmää, sillä muita vanhempia järjestelmiä ei ole tarkoituksenmukaista käsitellä, koska ne poikkeavat jo merkittävästi nykymuotoisista järjestelmistä.

Taulukko 4.1. *Järjestelmien yhteiset piirteet.*

Ominaisuus	Kuvaus	Lukumäärä
Tietoyhteys	Tarjoaa yhteyden tietokantaan	3
Käyttäjänhallinta	Käyttäjien tunnistus ja käyttöoikeudet	3
Suunnittelu	Gantt-suunnitelman käsittely	3
Optimointi	Automaattinen suunnitelman optimointi	2
Simulointi	Suunnitelman simulointi	1
Integrointi	Integrointi muihin tietojärjestelmiin	3
Raportointi	Raporttien tekeminen suunnitelmasta	2
Skenaarionhallinta	Eri suunnitelmanvaihtoehtojen hallinta	1
Yleiset ominaisuudet	Latausruudut ja ohjekentät	1
Parametrit	Asetukset ja suunnitteluparametrit	3
Tuotehallinta	Tuoterakenteet ja vaaditut resurssit	1
Linjahallinta	Tuotantopaikkojen ja –linjojen hallinta	1
Tulokset	Suunnitelman tarjoamat tulokset	2

Eräs ongelma järjestelmien yhteisten piirteiden suhteen on, että vaikka samoja ominaisuuksia onkin havaittavissa, niin varsinainen toteutus tai ominaisuuden käyttö saattaa poiketa hyvin paljon muiden järjestelmien vastaavista piirteistä. Lisäksi järjestelmät ovat asiakaskohtaisesti räätälöityjä, joten on vaikea ennustaa millaisia järjestelmiä tulevaisuudessa aiotaan toteuttaa. Ulkopuolisiin järjestelmiin tai asetuksiin liittyvät piirteet saattavat myös vaihdella, sillä esimerkiksi käyttäjänhallinta voidaan toteuttaa PES:n omaa tietokantaa käyttäen tai vaihtoehtoisesti suoraan Windowsin käyttäjätunnuksia hyödyntäen. Integraatiot vaihtelevat myös asiakaskohtaisesti – integrointeja ja tietoyhteyksiä on tehty muun muassa muihin tietokantoihin, XML- ja Excel-tiedostoihin. Tämän takia suunniteltavassa arkkitehtuurissa tulee huomioida mahdollisuus kyseisten ominaisuuksien toteuttamiseen ja vaihtelevuuteen. Tuoterunkoa rakenta-

essa tuleekin keskittyä huolellisesti tuotteiden yhteisiin ja eroaviin piirteisiin sekä määrittää tarkasti miten tuotteistus toteutetaan.

4.4 Muunneltavuus

Muunneltavuus eli varianssi ja sen hallinta on yksi tuoterunkojen keskeisimmistä ongelmista. Tuoterunkojen tuotteilla voi olla yhteisiä ja eroavia piirteitä, jolloin on tärkeää tunnistaa muunneltavuusvaatimukset. Suunniteltavan järjestelmän muunneltavuusvaatimuksia on esitetty taulukossa 4.2. Ominaisuudet ovat samat kuin taulukossa 4.1.

Taulukko 4.2. *Muunneltavuusvaatimukset.*

Ominaisuus	Variaatio
Tietoyhteys	• Microsoft SQL Server • Oracle RDBMS • XML-tiedostoluku
Käyttäjänhallinta	• Windows-käyttäjäryhmät • Käyttäjätiedot tietokannassa
Suunnittelu	• ComponentOne • WinChart
Optimointi	• Eri optimointialgoritmit
Simulointi	• Enterprise Dynamics • AnyLogic
Integrointi	• Microsoft Dynamics AX • SAP ERP • V10 toiminnanohjausratkaisut
Raportointi	• Raporttiformaatti: PDF, XML, Excel • Eri raporttimuodot ja –tiedot
Skenaarionhallinta	Ei muunneltavuusvaatimuksia
Yleiset ominaisuudet	Ei muunneltavuusvaatimuksia
Parametrit	Ei muunneltavuusvaatimuksia
Tuotehallinta	• Eri tuoterakenteet
Linjahallinta	Ei muunneltavuusvaatimuksia
Tulokset	• Tulosformaatti: PDF, XML, Excel • Eri tulosmuodot ja –tiedot

Monet muunneltavuusvaatimukset kohdistuvat ulkopuolisiin komponentteihin tai järjestelmiin ja ominaisuuksiin voi tulevaisuudessa kuulua myös muitakin muunneltavuusvaatimuksia, mutta niitä ei ole erikseen listattuna, vaan keskitytään jo olemassa olevien järjestelmien variaatioihin. Esimerkiksi tietoyhteys voi tulevaisuudessa asiakkaasta riippuen olla jokin muu kuin taulukossa 4.2 mainittu, ja näitä asioita pyritään huomioimaan arkkitehtuurisuunnitelmassa.

Tietokantana on käytetty Oraclen ja Microsoftin tietokantaratkaisuja, mutta myös XML-tiedostoja on osaltaan käytetty. Käyttäjänhallinnassa on usein luotu asiakkaalle omat käyttäjätunnukset, mutta on myös mahdollista käyttää automaattisesti Windowsin omia käyttäjäryhmiä, jolloin erillisiä käyttäjätunnuksia ei tarvita. Tuotantosuunnitelman laadinnassa käytetään apuna Ganttin kaavioita ja järjestelmässä Ganttin kaavi-

ot on toteutettu ulkopuolisilla komponenteilla: joko ComponentOnella tai WinChartilla. Simulointi suoritetaan vastaavasti ulkopuolisilla ohjelmilla: joko 4D Script –pohjaisella Enterprise Dynamicsilla tai Java-pohjaisella AnyLogicilla. Integrointi muihin järjestelmiin tarkoittaa käytännössä integrointia toiminnanohjausjärjestelmiin ja toistaiseksi käytettyjä toiminnanohjausjärjestelmiä ovat Microsoftin Dynamics AX, SAP:n ERP ja Logican V10. PES-tuotannonsuunnittelujärjestelmästä voidaan luoda myös erilaisia raportteja ja tulosnäkymiä, jotka ovat PDF-, XML- tai Excel-muodossa. Raporteissa ja tulosnäkymissä tulee huomioida, että eri asiakkailla on erilaiset tarpeet raporttien ja tulosten suhteen, kuten raporttien ja tulosten muoto sekä tietosisältö. Tuotehallinnassa tulee vastaavasti huomioida asiakkaan omat tuoterakenteet.

5 KOMPONENTTIARKKITEHTUURIN SUUNNITTELU

Luvussa 3 tutkittu nykyisen järjestelmän arkkitehtuuri ei mahdollista tehokasta uudelleenkäyttöä kohdassa 3.2 mainittujen ongelmakohtien takia. Tällöin on luontevaa suunnitella uudenlainen arkkitehtuuri, joka voisi vastata luvussa 4 esitettyihin vaatimuksiin ja haasteisiin. Uudelle arkkitehtuurille päätetään ensin keskeisimmät suunnitteluperiaatteet ja niiden avulla luodaan varsinainen arkkitehtuurisuunnitelma.

5.1 Yleiset suunnitteluperiaatteet

Ohjelmistosuunnittelussa on olemassa lukuisia hyväksi havaittuja suunnitteluperiaatteita. Osa noudatettavista suunnitteluperiaatteista saattaa olla miltei itsestäänselvyyksiä, mutta siitä huolimatta suunnitteluperiaatteet on hyvä dokumentoida, jotta kaikilla kehittäjillä on yhteinen näkemys. Nykyisen järjestelmän ongelmakohtien [kohta 3.2] ja suunniteltavan arkkitehtuurin vaatimusten [luku 4] perusteella noudatetaan erityisesti seuraavia suunnitteluperiaatteita:

- Single responsibility principle eli yhdellä komponentilla on vain yksi vastuualue. [kohta 2.1.3]
- YAGNI (”You ain’t gonna need it”) eli suunnitellaan vain tarpeelliset ominaisuudet ja vältetään tarpeetonta etukäteissuunnittelua, sillä suunnitelmat kehittyvät ja muuttuvat ajan myötä.
- Principle of least knowledge eli komponentti ei tiedä ulkopuolisista toteutusyksityiskohdista.
- Proper naming conventions eli käytetään yhtenäistä ja selkeää nimeämistapaa.

Luonnollisesti on olemassa lukuisia muitakin hyviä suunnittelusääntöjä, mutta olennaista on keskittyä ongelmakohtien ja vaatimusten kannalta keskeisimpiin osa-alueisiin.

5.2 Kerrosarkkitehtuuri

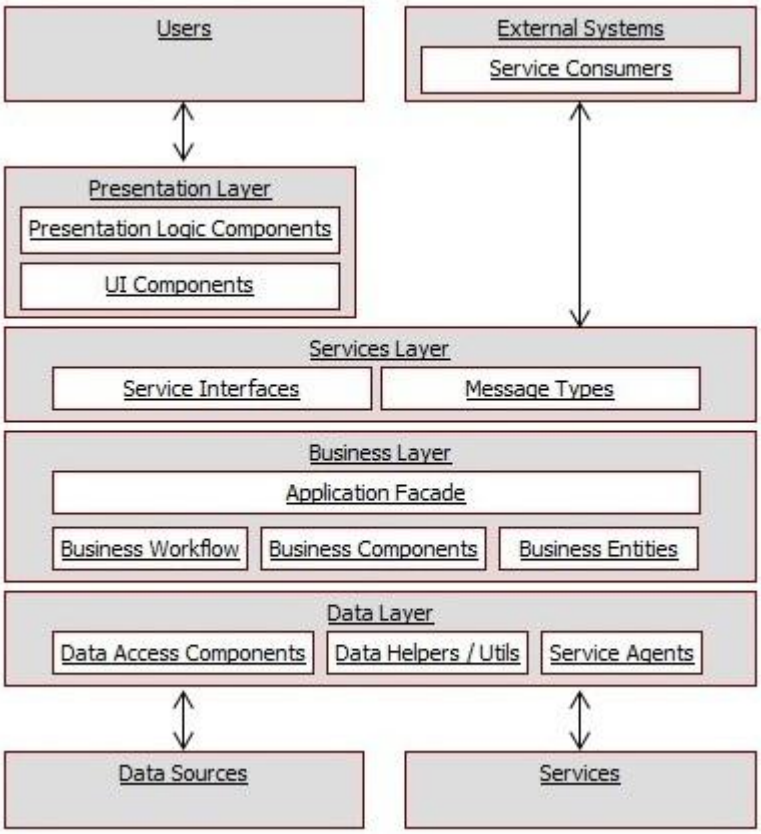
Kerrosarkkitehtuuri on hyvin tyypillinen tapa kuvata järjestelmiä. Kerrosarkkitehtuurilla voidaan luoda selkeä rakenne monenlaisille järjestelmille ja eri vastuualueet on helppo erottaa. Lisäksi kerrosarkkitehtuuri on intuitiivinen ja läheinen vanhoihin järjestelmiin.

5.2.1 .NET-kerrosarkkitehtuurityyli

Arkkitehtuurityylin vaatimuksena on, että tyyli ei saa olla liian kaukana olemassa olevien järjestelmien tyylistä. Lisäksi vaatimuksena on, että järjestelmät kehitetään Win-

dows-ympäristössä .NET-kirjaston avulla. [kohta 4.1] Tällöin luontevaksi arkkitehtuurityyliksi soveltuu Microsoft Application Architecture Guide –kirjan kuvaama kerrosarkkitehtuuri [kuva 5.1]. Kuvan 5.1 kerrosarkkitehtuuri on suunniteltavan arkkitehtuurin perusrakenne, jota sovelletaan organisaation tarpeiden ja asetettujen vaatimusten mukaisesti.

Kuvassa 5.1 varsinainen järjestelmä koostuu neljästä eri kerroksesta: Presentation Layer, Services Layer, Business Layer ja Data Layer. Presentation Layer (esityskerros) sisältää käyttäjäkeskeisen toiminnallisuuden ja Presentation Layer voidaan jakaa kahteen pääalueeseen: UI Components ja Presentation Logic Components. UI Components sisältää visuaaliset elementit tiedon esittämiseen ja vastaa käyttäjän syötteiden hallinnasta. Presentation Logic Components puolestaan sisältää käyttöliittymän varsinaisen sovelluskoodin, joka mahdollistaa loogisen toiminnan ja rakenteen. Tässä sovelluskoodissa on mahdollista soveltaa MVC-mallia tai jotain MVC-mallin johdannaista.



Kuva 5.1. .*NET-ympäristön tyypillinen sovellusarkkitehtuuri kerrosmallina.[13]*

Services Layer voidaan toteuttaa, mikäli havaitaan tarve ulkopuolisille järjestelmille. Tällöin on luontevaa, että ulkopuoliset järjestelmät käyttävät erillistä rajapintaa järjestelmien heterogeenisyyden vuoksi. On myös huomattavaa, että kuvan 5.1 arkkitehtuurityyli ei vaadi, että Presentation Layer kommunikoi ainoastaan Service Layerin kanssa, jolloin Presentation Layer voi olla suorassa yhteydessä Business Layeriin. Services Layer voidaan jakaa myös kahteen pääalueeseen: Service Interfaces ja Message Types. Service Interfaces on rajapinta, jota kaikki ulkopuoliset järjestelmät käyttävät (vertaa fasadi-suunnittelumalli) ja Message Types puolestaan vastaa tietytyyppien ja tietorakenteiden käärimisestä (wrapping).

Business Layer sisältää varsinaisen toimintalogiikan. Business Layer voidaan jakaa neljään osaan: Application Facade, Business Workflow, Business Components ja Business Entities. Application Facade on vastaavanlainen yksinkertaistettu rajapinta kuin Services Layerin Service Interfaces. Application Facade voi yhdistää useita operaatioita ja siten vähentää riippuvuuksia, sillä operaatioiden kutsujien ei tarvitse tietää toteutusyksityiskohdista. Business Components –osan vastuulla on sovellusdatan käsittely, käsittelysäännöt sekä validin sovellusdatan varmistaminen. Suotavaa on, että Business Components ei sisällä mihinkään yksittäiseen käyttötapaukseen liittyvää logiikkaa, jotta voidaan maksimoida uudelleenkäytettävyys. On tavallista, että käyttäjän antamat syötteet ja käskyt vaativat useampien operaatioiden yhdistämistä oikeassa järjestyksessä ja lisäksi eri komponentit saattavat toteuttaa eri operaatiota. Business Workflow vastaa, että eri komponenttien operaatiot tulee suoritettua oikeassa järjestyksessä, jotta koko käyttäjän antama komento voidaan suorittaa. Business Entities –alueen vastuulla on puolestaan entiteettien eli todellista maailmaa kuvaavien objektien (esimerkiksi tuote, toimipiste tai tilaus) logiikan ja datan kapseloinnista.

Data Layer mahdollistaa yhteyden järjestelmän ulkopuolisiin datalähteisiin (kuten tietokannat tai ulkopuoliset järjestelmät) ja osaltaan vastaa järjestelmän sisäisen datan hallinnasta. Data Layer voidaan jakaa kolmeen osa-alueeseen: Data Access Components, Data Helpers / Utilities ja Service Agents. Data Access Components abstrahoi logiikan, joka mahdollistaa yhteyden ulkopuolisiin datalähteisiin, jolloin olennainen toiminnallisuus tulee keskitettyä ja sovellusta on helpompi ylläpitää. Service Agents toteuttaa vastaavan idean kuin Data Access Components, mutta Service Agents on yhteydessä ulkopuolisiin järjestelmiin, jolloin semantiikka, kommunikointi ja ominaisuudet oletettavasti poikkeavat muun muassa tavanomaisemmasta tietokantayhteydestä. Data Helpers / Utilities sisältää Data Layerillä tarvittavia apuvälineitä ja työkaluja. [13, s. 19-134]

5.2.2 Suunniteltava kerrosrakenne

Suunniteltava kerrosrakenne perustuu kuvan 5.1 kerrosarkkitehtuurityyliin. Kerrosrakenteen suhteen tulee miettiä, että onko tavoitteena tehdä vain loogisia kerroksia vai mahdollistaa myös kerrosten eri fyysinen sijainti. Tärkeimpien laatuominaisuuksien (muunneltavuus ja ylläpidettävyys) korostamiseksi jokainen kerros pyritään pitämään mahdollisimman koherenttina ja heikosti sidottuna muihin kerroksiin, sillä on mahdollista, että tulevaisuudessa kerrokset sijaitsevat myös fyysisesti eri paikoissa. Näin ollen kaikkien kerrosten logiikka tulee abstrahoida. Abstrahointia voidaan toteuttaa monin eri tavoin, kuten abstrakteilla rajapinnoilla, viestinvälityksellä ja riippuvuusinversiolla (dependency inversion).

Lisäksi kerrosrakenteen kommunikointisäännöt tulee päättää tärkeimpien laatuominaisuuksien perusteella. Selkeästi tärkein laatuominaisuus on muunneltavuus. Käytännössä tämä tarkoittaa, että kerrosten vuorovaikutuksessa kerrosten ohitukset eivät ole missään tilanteessa sallittuja esimerkiksi suorituskyvyn parantamiseksi (strict interaction).

Kerrosrakenteen yksi keskeisin kysymys on, että mitä kerroksia tarvitaan. Yksikään aikaisempi järjestelmä ei käytä ulkopuolisia järjestelmiä hyväkseen [luku 3], joten voidaan päättää, että kuvassa 5.1 oleva Services Layer ei ole järjestelmän kannalta oleellinen. Lisäksi suunnitteluperiaatteissa [kohta 5.1] esitettiin, että ei suunnitella ominaisuuksia, joille ei vielä ole esiintynyt tarvetta. Muut kerrokset ovat perusteltuja vanhan järjestelmän rakenteen, suurien datamäärien ja osittain monimutkaisen toimintalogiikan perusteella.

5.2.3 Yksittäisten kerrosten kuvaukset

Suunniteltava järjestelmä on luonteeltaan rikas asiakassovellus (rich client application), joten on luontevaa erottaa varsinainen näkymä käyttöliittymälogiikasta. Tätä varten on kehitetty useita malleja, kuten MVC-malli. Käytettäväksi malliksi valitaan kuitenkin MVP-malli (Model-View-Presenter), jossa Presenter-osa on Controller-osan sijaan välittäjänä Model- ja View-osan välissä. MVP-mallissa Model- ja View-osat on siis erotettu toisistaan kokonaan, mikä osaltaan tukee paremmin vaatimuksissa määritettyä modulaarisuutta.

Toimintalogiikkakerroksen komponenttien tulee olla erittäin heikosti sidottuina muihin komponentteihin, sillä tavoitteena on osaltaan suunnitella runkoa useammalle järjestelmälle. Tällöin komponenttien tulee olla vaihdettavissa tai muunneltavissa ilman, että muu toimintalogiikka kärsii. Lisäksi on mahdollista, että komponentteja tulee myöhemmin lisää tai että komponentteja konfiguroidaan asiakaskohtaisesti.

Datalähteet ovat pääasiassa relaatiotietokantoja tai XML-dokumentteja [kohta 4.4], joten luonteva valinta on käyttää LINQ:a, sillä se mahdollistaa kyselyt niin relaatiotietokantoihin (LINQ to SQL) että XML-dokumentteihin (LINQ to XML) ja toisaalta myös moniin muihin dataformaatteihin. Muunneltavuuden ja ylläpidettävyyden maksimoimiseksi datakerroksella pyritään abstrahoimaan datayhteydet, -kyselyt ja -tyypit mahdollisimman hyvin.

5.3 Kerrosarkkitehtuurin rakennekuvaus

Arkkitehtuurisuunnitelma toteutetaan asetettujen vaatimusten ja suunnitteluperiaatteiden perusteella. Kuvauksessa käytetään vastaavia kaavioita kuin vanhan järjestelmän kuvauksessa [luku 3]. Lisäksi suunnitteluun on otettu mukaan komponenttikaavio, sillä keskeinen idea on juuri komponenttipohjainen kehitys. Arkkitehtuurikuvaksen lähtökohtana on kuvassa 5.1 esitetty .NET-kerrosarkkitehtuurityyli ja kohdassa 5.2 kuvattu kerrosarkkitehtuuri.

5.3.1 Fyysinen rakenne

Fyysinen rakenne noudattaa hyvin pitkälti kuvassa 3.1 esitettyä jaottelua. Tutkittava järjestelmä sekä vanhat järjestelmät noudattavat vastaavaa rakennetta, joten siihen ei ole tarpeellista tehdä muutoksia. Huomioitavaa kuitenkin on, että suunniteltavassa arkkiteh-

tuurissa eri kerrokset pyritään erottamaan toisistaan hyvin vahvasti, jolloin kerrosten sijoittaminen fyysisesti eri paikkoihin olisi tulevaisuudessa mahdollista. Nykymallisia järjestelmiä käytetään erilliseltä asiakaspäätteeltä esimerkiksi etäyhteydellä, mutta tulevaisuudessa on mahdollista, että näkymäkerros voisi sijaita myös asiakaspäätteellä. Vastaavasti datakerros voisi sijaita erillisellä laitteella esimerkiksi turvallisuussyistä.

Kuten kuvassa 5.1 on esitetty, niin palvelukerroksen avulla voidaan käyttää myös järjestelmän ulkopuolisia palveluita. Arkkitehtuurisuunnitelmassa pyritään korkeaan modulaarisuuteen, jolloin palvelukerroksen lisääminen jälkeenpäin olisi myös mahdollista. Järjestelmän fyysisesti ulkopuolisia palveluita voisi tulevaisuudessa olla esimerkiksi pilvipalvelut optimoinnin ja simuloinnin vaatimaa laskentatehoa varten.

Integroinnit muihin järjestelmiin on tarkoitus pitää myös hyvin modulaarisina. Vaikka järjestelmäintegrointeja tehdään, niin järjestelmien ei tulisi tietää toisista järjestelmistä mitään. Esimerkiksi toiminnanohjausjärjestelmästä saatavat tilaustiedot tulisi erottaa kyseisestä järjestelmästä tallentamalla tiedot tietokantaan tai erilliselle tiedostolle. Tällöin eri järjestelmien fyysinen sijainti ei aiheuta muutoksia järjestelmän arkkitehtuuriin.

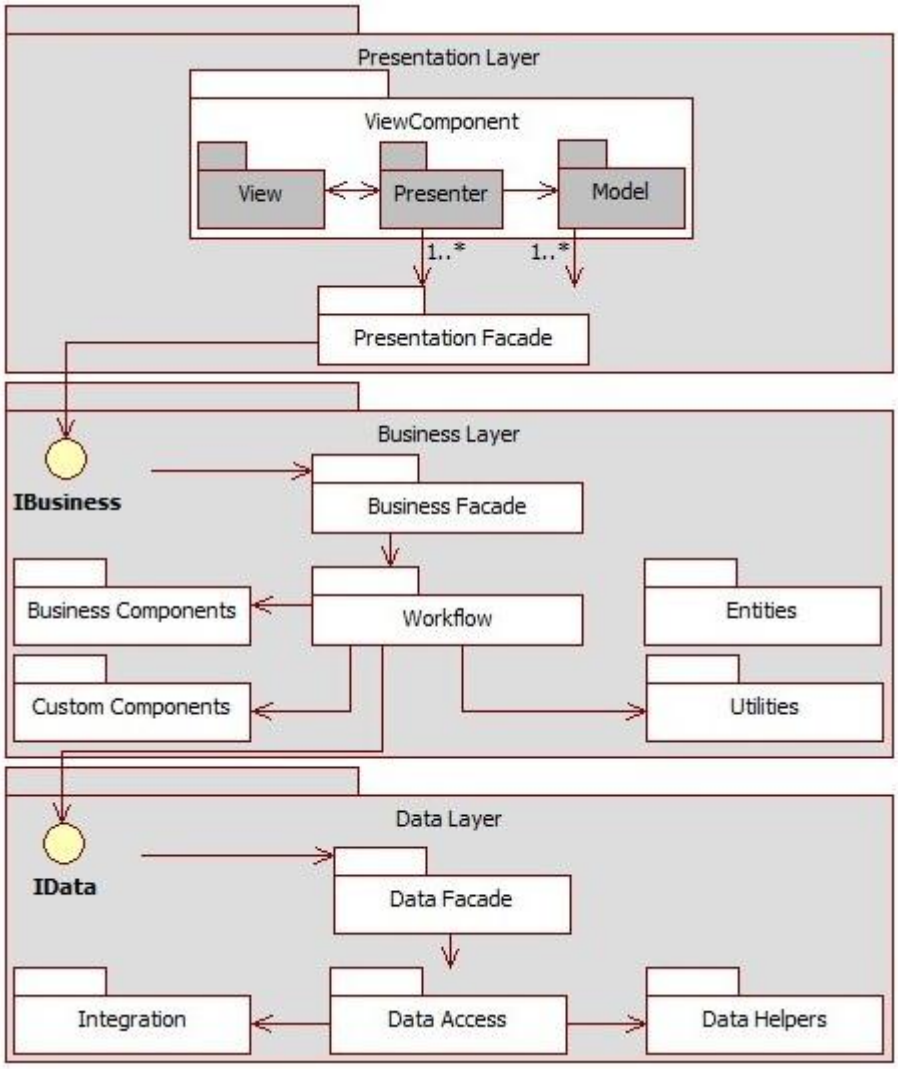
5.3.2 Looginen rakenne

Looginen rakenne koostuu kolmesta kerroksesta: esityskerros (Presentation Layer), toimintalogiikkakerroksesta (Business Layer) ja datakerroksesta (Data Layer) [kuva 5.2]. Kerrosten kommunikointi noudattaa suunnitteluperiaatteissa määriteltyä vuorovaikutustapaa eli niin kutsuttua strict interaction –tapaa, jossa ainoastaan ylemmät kerrokset voivat kutsua alempia ja kerroksia ei saa ohittaa. Lisäksi jokaisella alemmalla kerroksella on erillinen rajapinta, jota ylempi kerros käyttää, jolloin alemmalle kerrokselle on vain yksi liityntäkohta.

Esityskerroksella sovelletaan MVP-mallia [kohta 5.2.3]. ViewComponent-pakkauksia voi olla useita ja jokainen pakkaus sisältää MVP-mallin. MVP-mallissa View-osa sisältää varsinaisen käyttöliittymän, Presenter-osa sisältää käyttöliittymälogiikan ja Model-osa sisältää kyseisen käyttöliittymäkomponentin vaatiman datan. Tällöin samasta datasta voidaan tehdä erillisiä näkymiä ilman useita tietokantahakuja, mikä on todettu tarpeelliseksi myös vanhoissa järjestelmissä (esimerkiksi tiedon suodatuksessa eli filtteröinnissä). Käytännössä tämä tarkoittaa datan replikointia suoritustehon ja käytettävyyden parantamiseksi, jolloin myös tietokannan data lukitaan. Käyttöliittymän osat on jaettu eri ViewComponent-pakkauksen mukaisiin komponentteihin, jolloin yksittäinen komponentti on erotettu muista komponenteista ja komponenttia on mahdollisuus käyttää uudestaan.

Toimintalogiikkakerroksella järjestelmän toimintajärjestyslogiikka on sijoitettu Workflow-pakkaukseen. Kyseinen pakkaus vastaa, että halutun toiminnallisuuden vaatima komentosarja tulee suoritettua oikeassa järjestyksessä. Workflow-pakkaus voi käyttää hyödykseen erillisiä komponentteja (Business ja Custom Component –pakkaukset). Business Components –pakkaus ei sisällä mitään sovelluskohtaista logiikkaa, jotta kyseisiä komponentteja voisi käyttää uudelleen. Custom Components –pak-

kaukseen sen sijaan voidaan rakentaa sovelluskohtaisia komponentteja – tärkeintä on, että nämä ovat erillään ei-sovelluskohtaisesta toiminnallisuudesta. Utilities-pakkaus sisältää järjestelmän työkalut ja parametrit. Entities-pakkaus sisältää käytettävät entiteetit [kohta 5.2.1].



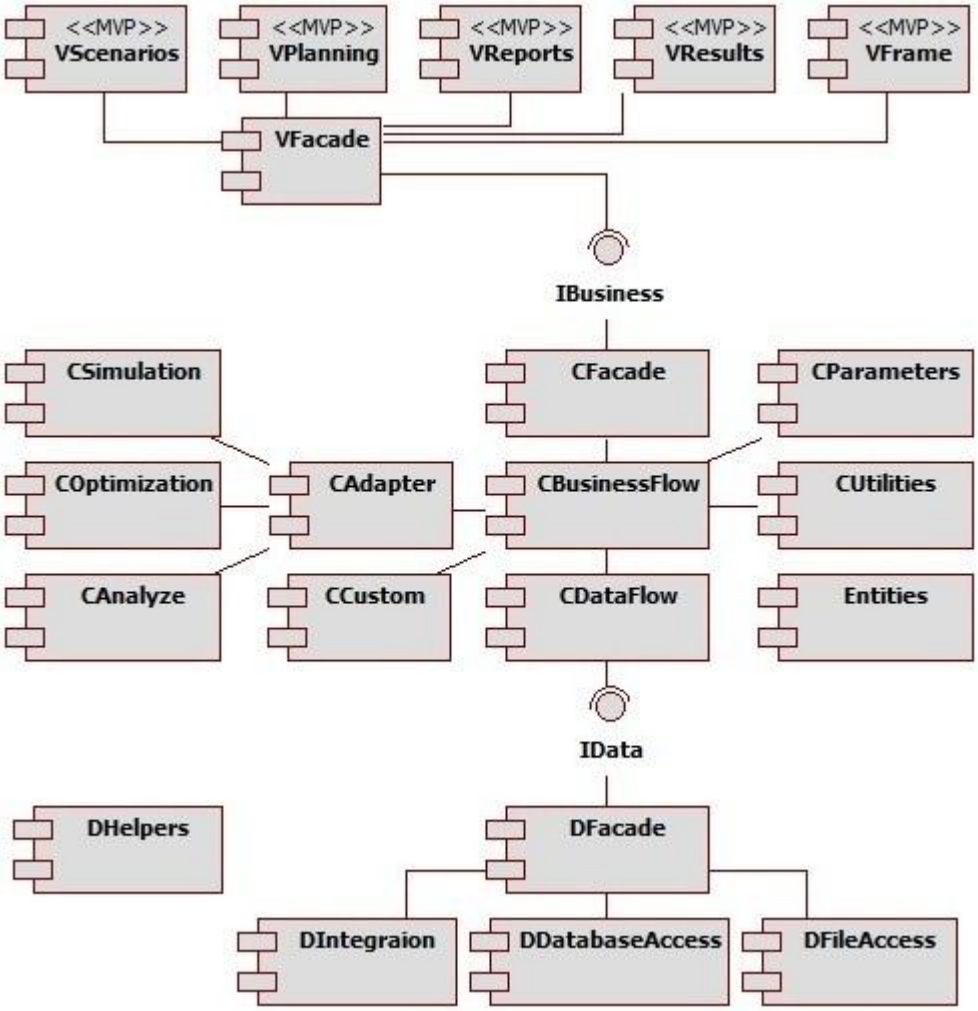
Kuva 5.2. *Suunniteltavan arkkitehtuurin pakkauskaavio.*

Datakerroksen Data Access –pakkaus vastaa tietoyhteyksistä niihin kohteisiin, joita järjestelmä käyttää (esimerkiksi tietokanta). Integration-pakkaus puolestaan vastaa järjestelmäintegroinneista ja niihin liittyvistä tietojenkäsittelystä, jotta ulkopuolelta saatu data on järjestelmän ymmärtämässä muodossa. Data Helpers –pakkaus sisältää Data Access –pakkauksen vaatimia apuominaisuuksia ja työkaluja esimerkiksi tiedon validointia varten.

5.3.3 Komponenttirakenne

Komponenttipohjainen kehitys on yksi järjestelmän keskeisimmistä tavoitteista ja tätä varten on luotu erillinen komponenttikaavio [kuva 5.3]. Ylimpänä on kuvattu esityskerroksen komponentit, jotka vastaavat suunniteltavan arkkitehtuurin käyttöliittymän päänäkymiä eli erillisiä ikkunoita käyttöliittymässä [liite 1]. Esimerkiksi VPlanning-

komponentti vastaa suunnittelunäkymästä. Jokainen näkymä on toteutettu erikseen omaksi komponentikseen, jotta samaa komponenttia on mahdollista käyttää uudestaan: komponentit eivät ole toisistaan riippuvaisia, ja uusia näkymäkomponentteja on mahdollista luoda. Vastaavasti esimerkiksi VPlanning-komponentti voi MVP-mallin avulla luoda erillisiä suunnittelunäkymiä Model-osaansa käyttäen. Komponentit voivat sisältää tai käyttää myös esimerkiksi käyttöliittymäkomponentteja (esimerkiksi toolbox- eli työkalukomponentit tai kolmannen osapuolen komponentit kuten Gantt-komponentti).



Kuva 5.3. *Suunniteltavan arkkitehtuurin komponenttikaavio.*

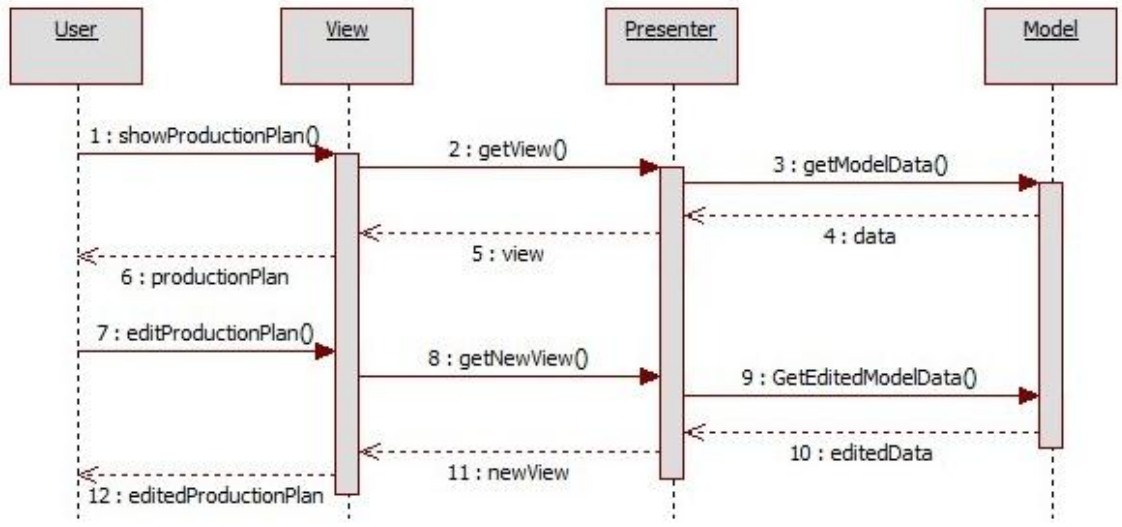
Keskellä on toimintalogiikkakerroksen komponentit, joiden nimeäminen aloitetaan C-kirjaimelle viitaten Control-sanaan. Toimintalogiikkakerroksen rajapinnan tarjoaa CFacade-komponentti ja eri komentojen ja operaatioiden suorittamisjärjestyksestä vastaa CBusinessFlow-komponentti. Mikäli toimintoa suoritettaessa on tarpeellista tehdä datakyselyjä, niin CDataFlow-komponentti vastaa dataoperaatioiden suoritusjärjestyksestä. Varsinaiset toimintalogiikkakomponentit (CSimulation, COptimization, CAnalyze) on yhdistetty CBusinessFlow-komponenttiin adapterin kautta (CAdapter). Tällä pyritään parantamaan uudelleenkäyttöä eri järjestelmien välillä tilanteissa, joissa käytettävää komponenttia tai suoritusjärjestyksestä vastaavaa komponenttia muutetaan. Adapterin tehtävänä on toimia sovittimena näiden komponenttien välillä ja eristää kyseisten

komponenttien suora yhteys toisiinsa. Huomattavaa on, että uusia komponentteja voidaan myös liittää CAdapter-komponentin rajapintaan. Kuten kohdassa 5.2.1 on mainittu, niin toimintalogiikkakomponentit eivät saa sisältää käyttötapaus- tai sovelluskohtaista toiminnallisuutta, sillä se heikentää uudelleenkäytettävyyttä. Järjestelmäkohtaisia toimintalogiikkakomponentteja on mahdollista käyttää, mutta ne tulee erottaa, kuten kaaviossa CCustom-komponentille on tehty.

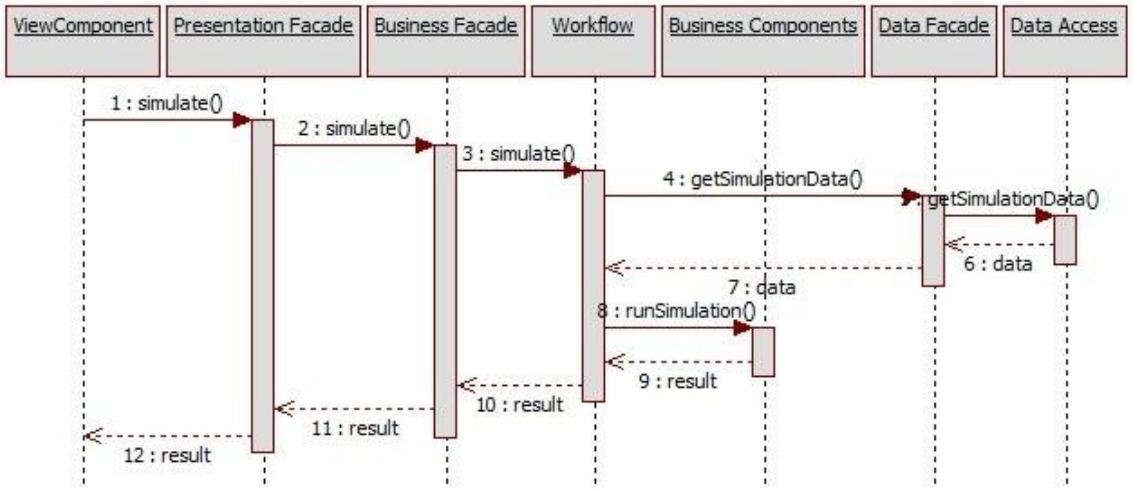
Alimpana on datakerros, jolla on vastaavanlainen rajapintarakenne kuin toimintalogiikkakerroksella. Datakerroksella on kolme pääkomponenttia, jotka vastaavat tietoyhteyksistä. DDatabaseAccess-komponentti vastaa nimensä mukaisesti tietokantayhteyksistä ja DFileAccess-komponentti puolestaan muun muassa XML-tiedostojen lukemisesta järjestelmään. Datakerroksella on myös integraatiorajapinta (DIntegration) ulkopuolisille järjestelmille. DIntegration-komponentti voi esimerkiksi suorittaa transaktioita ulkoisten järjestelmien tai niiden tietokantojen kanssa. Mikäli myöhemmin tulee tarvetta jollekin uudelle tietoyhteydelle, niin se on mahdollista liittää datakerrokselle muiden komponenttien tavoin.

5.3.4 Järjestelmän vuorovaikutus

Vuorovaikutus on pitkälti kerrosarkkitehtuurille tyypillistä kutsujen välitystä toisille järjestelmän osille. Kuvassa 5.4 on esitetty näkymätason vuorovaikutus MVP-mallin mukaisesti. Kuvassa 5.4 käyttäjä haluaa nähdä tuotantosuunnitelman, joka on tässä tapauksessa ladattu valmiiksi Model-osaan. View-osa rekisteröi kutsun ja pyytää Presenter-osalta sopivaa näkymää. Presenter-osa hakee tuotantosuunnitelman datan Model-osalta ja kutsuketju palautuu käyttäjälle. Käyttäjä haluaa lisäksi muokata suunnitelmaa, jolloin vastaava kutsuketju kulkeutuu malliin asti, johon muutokset päivitetään. Lopuksi muutetun tuotantosuunnitelman data palautuu käyttäjälle. Etuna tässä on, että mallin muutokset voidaan kaikki rekisteröidä näkymätasolla, ja varsinaiset tallennukset tietokantaan voidaan toteuttaa erikseen.



Kuva 5.4. *MVP-mallin toiminta näkymätasolla*.



Kuva 5.5. *Simuloinnin suoritusjärjestys eri komponenttien välillä.*

Kuvassa 5.5 on puolestaan esitetty simuloinnin suoritusjärjestys eri komponenttien välillä. Kutsuketju käy kaikilla järjestelmän tasoilla. Vuorovaikutus alkaa näkymätasolta, josta tulee toimintalogiikkakerrokselle pyyntö suorittaa simulointi. Workflow-osa suorittaa tarvittavien kutsujen koordinoinnin hakemalla ensin tarvittavan datan datakerrokselta ja tämän jälkeen käyttää simulointikomponenttia varsinaisen simuloinnin suorittamiseen. Simulointikomponentti palauttaa simulointitulokset Workflow-osalle, josta tulokset välitetään edelleen näkymätasolle asti.

Sekvenssikaavioista voidaan havaita suuri määrä kutsujen ja vastausten välitystä eri osien välillä. Tämä tavallisesti hidastaa suoritusta, mutta toisaalta parantaa modulaarisuutta, joka on huomattavasti tärkeämpi laatuvaatimus [kohta 4.1].

6 RATKAISUN ARVIOINTI

Arkkitehtuuria voidaan arvioida systemaattisesti käyttäen arviointiin soveltuvaa menetelmää. Erilaisia menetelmiä on olemassa useita, mutta tässä työssä keskitytään ATAM:iin (architecture trade-off analysys method).

6.1 Arkkitehtuurin arviointi

Ennen ATAM:in käsittelyä keskitytään arkkitehtuurin arviointiin yleisesti vastaamalla lyhyesti seuraaviin kysymyksiin:

- Miksi arkkitehtuuria pitäisi arvioida ja mitä hyötyä siitä on?
- Milloin arkkitehtuuri tulisi arvioida?
- Mitä haittapuolia tai vaikeuksia arvioinnissa on?

Arkkitehtuurin arvioinnille on useita perusteluita. Keskeisin syy on, että arkkitehtuurilla on todella suuri merkitys muun ohjelmiston kannalta, joten arkkitehtuuriin kannattaa panostaa. Arvioinnilla voidaan todeta erilaisten laatuvaatimusten täyttäminen. Tyypillisiä laatuominaisuuksia ovat esimerkiksi muunneltavuus, ylläpidettävyys, suorituskyky, turvallisuus, saatavuus ja siirrettävyys. Nämä laatuominaisuudet tulee myös käsiteltyä eri sidosryhmien näkökulmasta, jolloin arkkitehtuuri ei ole vain ohjelmistoarkkitehtien varassa. Tällöin myös eri sidosryhmät, kuten myyntiosasto, asiakkaat ja johtoryhmä, saavat paremman käsityksen arkkitehtuurista. Arviointi toimii siis myös kommunikoinnin edistäjänä tiedonvälityksen ja yhteisen sanaston ansiosta. Lisäksi arviointi vaatii suunnittelijoita ymmärtämään ja perustelemaan suunnitteluratkaisut.

Arviointi tulisi suorittaa kehityksen alkuvaiheessa, koska silloin ongelmia ja vikoja on helpompi ratkoa ja korjata. Tällöin arvioinnin avulla voidaan saada suurta taloudellista etua säästettyjen työtuntien ansiosta. Aivan alkuvaiheessa arviointi voi olla kevyempi ja laajamittainen arviointi voidaan toteuttaa vasta kun koko arkkitehtuuri on valmis. Toisaalta arvioinnista on hyötyä myös jo valmiille järjestelmälle erityisesti yhteisymmärryksen ja tiedonvälityksen takia. Bass [33] esittää, että arkkitehtuuriarvioinnin tulisi olla yksi ohjelmistokehityksen perusosista.

Arvioinnin haittapuolina voidaan mainita sen raskaus. Täysimittaiset arviointisessiot vaativat paljon resursseja: aikaa ja henkilöstöä. Tämän takia arviointeja sovelletaankin omalle organisaatiolle sopivaan muotoon. Arviointi voi olla myös arkkitehdeille hieman tarpeeton, koska arkkitehdit saattavat hyvinkin olla perillä suunnitteluratkaisujen laadusta ja arkkitehtuurin vahvuuksista sekä heikkouksista. Tällöinkin arviointi on kuitenkin hyödyllinen kommunikoinnin edistäjänä. [4, luku 9; 33, s. 261-305]

6.2 Architecture Trade-off Analysis Method

ATAM edellyttää tiettyjä osallistujia arviointiprosessiin. Nämä osallistujat ovat evaluointiryhmä, päättäjät ja vastuuhenkilöt. Evaluointiryhmä koostuu kolmesta viiteen arkkitehtuurin vastuuryhmän ulkopuolisista henkilöistä ja jokaiselle henkilölle on asetettu yksi tai useampi rooli kuten ryhmänjohtaja, ajanhallitsija tai skenaariokirjuri. Päättäjät ovat henkilöitä, joilla on valta vaikuttaa koko projektiin. Tähän ryhmään kuuluu projektipäälliköt, asiakkaat ja pääarkkitehti. Vastuuhenkilöiden ryhmä muodostavat ne henkilöt, jotka toteuttavat arkkitehtuurin. Tähän ryhmään kuuluu siis muun muassa suunnittelijat, testaajat, integraattorit ja ylläpitäjät.

ATAM-pohjaisen arviointiprosessin lopputuloksena saavutetaan ytimekäs arkkitehtuurikuvaus, laatuvaatimukset, herkkyys- ja tasapainokohdat, riskit ja turvalliset ratkaisut sekä käsitys arkkitehtuurin ja liiketoimintatavoitteiden yhdistämisestä. Näiden perusteella arkkitehtuuria voidaan kehittää paremmaksi. Lopputulosten saavuttaminen edellyttää siis tietyt osallistujat sekä ATAM-prosessin, johon perehdytään tarkemmin tässä alaluvussa. ATAM-prosessi voidaan jakaa neljään vaiheeseen ja yhdeksään askeleeseen. [33, s. 272-275]

6.2.1 Esittely

Esittelyvaihe sisältää kolme askelta (askeleet 1-3). Ensimmäisessä askeleessa arvioinnin johtaja esittelee ATAM-prosessin kaikille osallistujille. Tässä askeleessa kuvataan arviointiprosessi, roolit, tekniikat, tulokset sekä selvitetään mitä arviointiprosessilta voidaan odottaa.

Toisessa askeleessa käsitellään järjestelmää liiketoiminnan kannalta. Liiketoimintatavoitteet esittelee tavallisesti projektipäällikkö tai asiakas. Keskeisiä asioita ovat järjestelmän toiminnot, rajoitteet (kuten tekniset, taloudelliset tai poliittiset rajoitteet), sidosryhmät ja järjestelmän liiketoimintatavoitteet.

Kolmannessa askeleessa pääarkkitehti kuvaa järjestelmän arkkitehtuurin sopivalla tarkkuustasolla (sopiva tarkkuustaso vaihtelee esimerkiksi käytettävien resurssien ja järjestelmän koon suhteen). Arkkitehti kuvaa muun muassa teknisen toimintaympäristön, järjestelmän vuorovaikutussuhteet ja rajapinnat. Esittelyssä on hyvä käyttää apuna esimerkiksi käsite-, luokka-, sijoittelu- ja komponenttikaavioita. Hyvä sääntö on valita ne kaaviot, jotka arkkitehti kokee tärkeimmiksi arkkitehtuurin rakennusvaiheessa. [33, s. 276-279]

6.2.2 Analyysi

Analyysivaihe sisältää myös kolme askelta (askeleet 4-6). Neljännessä askeleessa käydään läpi keskeisimmät arkkitehtuuriratkaisut sekä niiden vaikutukset laatuominaisuuksiin. Keskeisimpiä asioita ovat arkkitehtuurityylit, suunnittelumallit ja menetelmät. Näihin asioihin tulee myös liittää selvitykset, miten ratkaisu vaikuttaa laatuominaisuuksiin ja miksi kyseistä ratkaisua on käytetty.

Viidennessä askeleessa laaditaan laatupuu. Laatupuu sisältää hierarkisessa järjestyksessä korkean tason laatuominaisuudet, täsmennetyt laatupiirteet sekä esimerkkitilanteita [kuva 6.1]. Jokaiselle laatupiirteelle annetaan kaksi parametria: ensimmäinen parametri kertoo kuinka tärkeä kyseinen piirre on ja toinen parametri kertoo kuinka paljon resursseja piirteen täyttäminen vaatii. Näissä parametreissa voidaan käyttää tavallisesti kolmea arvoa, kuten low, medium ja high tai asteikkoa 1-3.

Kuva 6.1. *Yksinkertaistettu esimerkki laatupuusta.*

Kuudennessa askeleessa yhdistetään laatupuu ja arkkitehtuuriratkaisujen kuvaukset. Kuudennen askeleen tavoite on tunnistaa riskit, turvalliset ratkaisut, herkkyyskohdat ja tasapainokohdat. Riski on suunnitteluratkaisu, joka voi johtaa jonkin laatuominaisuuden heikkenemiseen. Riskistä on tärkeää kuvata kyseinen ratkaisu, ratkaisun aiheuttama ongelma sekä ongelman syy. Turvallinen ratkaisu puolestaan parantaa jotain laatuominaisuutta. Myös turvallinen ratkaisu dokumentoidaan kuvaamalla itse ratkaisu, sen tuomat edut ja syyt. Herkkyyskohta on jonkin laatuominaisuuden kannalta kriittinen ratkaisu. Jos ratkaisua muutetaan, niin jokin laatuominaisuus oletettavasti heikentyy. Tasapainokohta (tradeoff point) on herkkyyskohta, joka vaikuttaa kahteen tai useampaan laatuominaisuuteen. Tasapainokohdalle tyypillistä on, että se parantaa jotain laatuominaisuutta ja heikentää jotain toista (esimerkiksi parantaa muunneltavuutta, mutta heikentää suorituskykyä). [4, s. 229-232; 33, s. 279-284]

6.2.3 Testaus

Testausosio sisältää kaksi askelta (askeleet 7-8). Seitsemännessä askeleessa eri osapuolet pohtivat yhdessä skenaarioita omasta lähtökohdastaan. Asiakkaat voivat esimerkiksi keksiä tiettyjä käyttötapauksia ja toisaalta ylläpito- ja käyttäjätukiryhmä voi miettiä erilaisia muutos- ja ylläpitoskenaarioita. Näin löydettyjä skenaarioita voidaan verrata aikaisemmin tehtyyn laatupuuhun, jotta voidaan käsitellä arkkitehtuuria laajemmasta näkökulmasta. Jos vastaava skenaario löytyy laatupuusta, niin kyseinen skenaario on jo huomioitu suunnitelmassa. Jos vastaavaa skenaariota ei löydy, niin uusi skenaario voidaan liittää uudeksi lehdeksi. Jos vastaavaa laatuominaisuutta ei löydy ollenkaan, niin kyseessä on joko laatuominaisuuksiin liittymätön skenaario tai sitten kokonaan uusi laatuominaisuus. Kaikkiaan tavoitteena on testata arkkitehtuuria laajemmalla sidosryhmäkokonaisuudella.

Kahdeksannessa askeleessa uudet skenaariot käsitellään arkkitehtuurin kannalta. Tällöin arkkitehti voi esitellä mahdollisesti uusia arkkitehtuuriratkaisuja tai ainakin kertoa kuinka eri skenaariot toteutuvat arkkitehtuurilla. [4, s. 232-233; 33, s. 300-302]

6.2.4 Raportointi

Viimeisessä askeleessa esitetään saadut tulokset kaikille arviointiin osallistuneille osapuolille noin 1-2 tuntia kestävässä esitelmätilaisuudessa. Raportointivaiheen keskeinen idea on tuoda esiin arkkitehtuurissa käytetyt ratkaisumallit, jotka vaikuttavat laatuominaisuuksiin. Tällöin voidaan tunnistaa arkkitehtuurin vahvuuksia ja heikkouksia sekä ymmärtää paremmin arkkitehtuuria kokonaisuutena.

Tulokset tulee esittää systemaattisesti. Tuloksia on hyvä käsitellä esimerkiksi skenaarioittain, jolloin esitellään skenaarion toteuttava ratkaisu, vaikuttavat laatuominaisuudet sekä mahdolliset riskit, herkkyyskohdat, turvalliset ratkaisut ja tasapainokohdat. Luonnollisesti muitakin lähestymistapoja voidaan käyttää – tärkeintä on, että ATAM palvelee toteuttavaa organisaatiota mahdollisimman hyvin. Lisäksi on syytä huomioida, että arkkitehtuurin korjaaminen ei varsinaisesti kuulu ATAM-prosessiin, vaan ATAM:in avulla voidaan arvioida ja testata arkkitehtuuria. [4, s. 233-234; 33, s. 302-303]

6.3 Suppea ATAM-prosessi

Kuten kohdassa 6.1 todettiin, täysimittainen ATAM-prosessi vaatii paljon aikaa ja henkilöstä, minkä takia on tavallista, että organisaatio toteuttaa kevyemmän version ATAM-prosessista. Tämän työn osalta ei kuitenkaan ole tarkoituksenmukaista toteuttaa ATAM-prosessia yhdessä muiden osastojen henkilöjen kanssa, vaan sen sijaan suorittaa arviointiprosessi, jonka perusteella arkkitehtuuri itse voi osaltaan varmistua suunnitelman laadusta ennen suunnitelman esittämistä laajemmalle yleisölle.

Suppea ATAM-prosessi hyödyntää kuitekin täysimittaisen ATAM-prosessin piirteitä. Suppea ATAM-prosessi voidaan jakaa neljään vaiheeseen. Ensimmäisessä vaiheessa laaditaan laatupuu ja skenaariot. Tähän vaiheeseen voidaan ottaa mukaan muita suunnitteluun läheisesti liittyviä henkilöitä kattavuuden parantamiseksi. Laatimisprosessi on kuitenkin pystyttävä pitämään riittävän nopeana, jotta suppea ATAM-prosessi ei menetä tarkoitustaan. Toisessa vaiheessa eri skenaariot pisteytetään tärkeyden ja vaikeuden suhteen. Skenaarion tärkeys on hyvä arvioida yhdessä sellaisen henkilön kanssa, jolla on selkeä käsitys vaadittavista ominaisuuksista. Asiakasrajapintaan arviointia ei kuitenkaan ole syytä viedä ellei asiakas ole erittäin läheinen tai helposti tavoitettavissa. Kolmannessa vaiheessa valitaan analysoitavat skenaariot esimerkiksi pisteytyksen ja tärkeimpien laatuominaisuuksien mukaan. Kun skenaariot on valittu, toteutetaan jokaisesta skenaariosta lyhyt analyysi, josta tulee ainakin ilmetä skenaarion kannalta keskeisimmät suunnitteluratkaisut sekä niiden muodostamat tasapainopisteet, riskit ja ei-riskit. Neljännessä vaiheessa toteutetaan arvioinnin yhteenveto ja päätetään mahdolliset muutokset arkkitehtuurisuunnitelmaan.

6.4 Arvioinnin toteutus

Tässä työssä arkkitehtuurin arviointia ei toteuteta täysimittaisena ATAM-prosessina, vaan suppeampana versiona [kohta 6.3], jossa luodaan järjestelmän laatupuu [liite 2] ja analysoidaan tärkeimmät skenaariot. Tämän työn laatupuu, skenaariot ja painotus on laadittu yhdessä Mikko Kunnarin kanssa, jotta lopputulos olisi kattavampi ja monipuolisempi. [34] Arvioitaviksi skenaarioiksi valitaan sellaiset skenaariot, jotka ovat sekä tärkeitä että vaikeita. Lisäksi skenaarioihin on valittu tärkeimmistä laatuominaisuuksista – ylläpidettävyys ja muunneltavuus – kummastakin yksi skenaario. Ensimmäinen skenaario on esitelty taulukossa 6.1.

Taulukko 6.1: *Skenaario 1.*

Skenaario	Ohjelman kaatuu odottamattomasta syystä, mutta käyttäjän muokkaamaa tai luomaa data ei kuitenkaan katoa		
Laatuominaisuus	Luotettavuus		
Ympäristö	Vikatilanne		
Ärsyke	Järjestelmän kaatuminen		
Vaste	Järjestelmä ei kadota dataa		
Arkkitehtuuripäätös	**Tasapainopiste**	**Riski**	**Ei-riski**
Datan replikointi näkymätasolle	T1		
Datakerroksen erottaminen			E1

Ohjelman kaatuessa käyttäjän data ei saisi kadota vaikka järjestelmä kaatuisi. Tämä tilanne voidaan hallita tallentamalla järjestelmän data tietokantaan riittävän usein (tarvittaessa jokaisella muokkauksella). Tämä kuitenkin heikentää suorituskykyä, minkä takia dataa replikoidaan näkymätasolle MVP-mallia hyödyntäen. Näin ollen datan replikointi muodostaa tasapainopisteen T1. Toisaalta, jos järjestelmän kaatuessa voidaan ongelmatilanne hallita suorittamalla tallennus järjestelmän kaatumisesta huolimatta, niin skeenario ei koidu ongelmaksi. Kaikissa mahdollisissa kaatumiseen johtavissa ongelmatilanteissa tämä ei kuitenkaan ole välttämättä mahdollista. Datakerroksen erottaminen on kuitenkin turvallinen ratkaisu, sillä se edistää modulaarisuutta ja erottaa asiakaspäätteen tietokannasta. Näin ollen datakerroksen erottaminen muodostaa turvallisen ratkaisun E1. Skenaario 2 on esitetty taulukossa 6.2.

Taulukko 6.2: *Skenaario 2.*

Skenaario	Kesätyöntekijä käyttää järjestelmää ja muuttaa vahingossa tuotantosuunnitelmaa		
Laatuominaisuus	Turvallisuus		
Ympäristö	Normaali toiminta		
Ärsyke	Käyttäjä, jolla ei ole käyttöoikeutta, muuttaa tuotantosuunnitelmaa		
Vaste	Tuotantosuunnitelma tulisi voida palauttaa		
Arkkitehtuuripäätös	**Tasapainopiste**	**Riski**	**Ei-riski**
MVP-malli			E2
Yhteinen datakerros	T2		

Luonnollisesti käyttäjän tunnistaminen ja käyttäjäoikeuksien rajaaminen on mahdollista, mutta skenaarion 2 tilanteessa kesätyöntekijä on luvatta mennyt käyttämään järjestelmää esimiehen työkoneelta. Tällaiseen tilanteeseen voidaan varautua muun muassa ky-

symällä salasanaa ennen tallennusta, mutta toisaalta jatkuva salasanan kyseleminen voi vähentää käyttömukavuutta. Toisaalta, jos kyseessä on vahinko, niin MVP-mallin avulla voidaan toteuttaa helposti uusia näkymiä esimerkiksi tallennuksen varmistamiseksi, jolloin MVP-malli luo turvallisen ratkaisun E2. Luonnollisesti vanha tuotantosuunnitelma tulisi voida palauttaa ja se on mahdollista, jos tietokantaan tallennetaan esimerkiksi tietty määrä tuotantosuunnitelmien vanhempia revisioita. On kuitenkin mahdollista, että virheellistä tuotantosuunnitelmaa toteutetaan jossain muualla yhteisen datakerroksen takia. Yhteinen datakerros on kuitenkin käytettävyyden ja saatavuuden kannalta keskeinen, mutta siitä huolimatta tässä skenaariossa yhteinen datakerros luo tasapainopisteen T2. Skenaario 3 on esitetty taulukossa 6.3

Taulukko 6.3: *Skenaario 3.*

Skenaario	Asiakas haluaa itse luoda uudet käyttäjäoikeudet työnjohtajille ja toimitusjohtajalle		
Laatuominaisuus	Varioitavuus		
Ympäristö	Normaali toiminta		
Ärsyke	Asiakas haluaa itse asettaa käyttöoikeudet		
Vaste	Asiakkaan asettamat käyttöoikeudet tulevat voimaan		
Arkkitehtuuripäätös	**Tasapainopiste**	**Riski**	**Ei-riski**
MVP-malli			E3

Järjestelmän toimittaja voi luonnollisesti varioida tuotteita liittämällä järjestelmään esimerkiksi eri komponentteja. Tämän lisäksi käyttäjän tulisi myös voida varioida tuotetta muun muassa käyttöoikeuksien suhteen. Tämä onnistuu MVP-mallia käyttäen, sillä MVP-mallin avulla voidaan luoda erilaisia näkymiä riippuen käytettävästä mallista. Malliin voidaan siis liittää esimerkiksi tieto käyttäjäoikeuksista, jolloin eri käyttäjäoikeuksilla luodaan erilaisia näkymiä. Tässä skenaariossa MVP-malli on siis turvallinen ratkaisu E3. Skenaario 4 on kuvattu taulukossa 6.4.

Taulukko 6.4: *Skenaario 4.*

Skenaario	Järjestelmään toteutetaan uusi optimointikomponentti		
Laatuominaisuus	Ylläpidettävyys		
Ympäristö	Järjestelmän kehitys		
Ärsyke	Vanha komponentti tulee vaihtaa uuteen		
Vaste	Uusi komponentti tulee käyttöön		
Arkkitehtuuripäätös	**Tasapainopiste**	**Riski**	**Ei-riski**
Komponenttijako			E4
Kerrosten eristäminen toisistaan	T4		

Jos järjestelmään toteutetaan uusi optimointikomponentti, joka käyttää esimerkiksi erilaista optimointialgoritmia, niin uusi komponentti tulisi voida vaihtaa vanhan tilalle. Tällöin komponenttijako on turvallinen arkkitehtuuriratkaisu (E4), sillä komponentin vastuualue ja rajapinnat eivät muutu. Sen sijaan kerrosten eristäminen luo tasapainopisteen T4, koska kerrosten eristäminen luonnollisesti parantaa modulaarisuutta, mutta toisaalta uuden komponentin testaaminen on vaikeampaa. Koko järjestelmä tulisi olla käytössä ja uusi komponentti tulisi voida testata asiakkaan omalla datalla. Jos kerrokset ovat fyysisesti eri paikoissa, niin asiakkaan järjestelmiä vastaavan ympäristön toteuttaminen voi olla hankalaa. Skenaario 5 on kuvattu taulukossa 6.5.

Taulukko 6.5: *Skenaario 5.*

Skenaario	Asiakkaan järjestelmässä ei ole simulointimahdollisuutta, mutta asiakas haluaa ostaa ominaisuuden järjestelmän toimituksen jälkeen
Laatuominaisuus	Muunneltavuus
Ympäristö	Järjestelmän kehitys
Ärsyke	Järjestelmään halutaan uusi ominaisuus
Vaste	Uuden ominaisuuden toiminnallisuus liitetään osaksi järjestelmää

Arkkitehtuuripäätös	Tasapainopiste	Riski	Ei-riski
Adapteri			E5
MVP-malli			E6
Järjestelmän osien tiukka vuorovaikutus	T5		

On mahdollista, että asiakas haluaa järjestelmän toimituksen jälkeen ostaa järjestelmään lisäominaisuuksia. Eräs lisäominaisuus voisi olla tuotannon simulointimahdollisuus. Tällöin järjestelmään liitetään simulointikomponentti ja liittämiseen käytetään avuksi adapteri-suunnittelumallia. Tällöin uusi komponentti saadaan eriytetyksi muusta järjestelmästä eikä siten vaikuta muiden osien toimintaan, jolloin adapterin käyttö luo turvallisen ratkaisun E5. Lisäksi MVP-mallin avulla voidaan luoda uusia näkymiä simulointia varten, jolloin kyseinen päätös luo turvallisen ratkaisun E6. Järjestelmän osien tiukkaa vuorovaikutusta voidaan pitää tasapainopisteenä T5, sillä simulointikomponentin lisääminen edellyttää muutoksia kaikille kerroksille ja muun muassa Workflow-komponenttiin tulee lisätä simulointikäskyn komennot. Toisaalta järjestelmän osien tiukka vuorovaikutus parantaa modulaarisuutta merkittävästi ja modulaarisuus voidaan oletettavasti arvostaa korkeammalle.

Taulukko 6.6: *Skenaario 6.*

Skenaario	Useat ihmiset haluavat käsitellä samaa dataa rinnakkain.
Laatuominaisuus	Saatavuus
Ympäristö	Normaali toiminta
Ärsyke	Useampi henkilö käyttää järjestelmää rinnakkain
Vaste	Järjestelmä mahdollistaa rinnakkaisen toiminnan

Arkkitehtuuripäätös	Tasapainopiste	Riski	Ei-riski
Tiedon replikointi		R1	
Tiedon lukitseminen	T6		

Asiakkaan henkilökunta haluaa mahdollisesti käyttää järjestelmää rinnakkain esimerkiksi eri toimipisteillä. Tehokkuussyistä tietoa replikoidaan näkymätason malliin, mutta rinnakkaiskäytössä tämä aiheuttaa riskin R1. Jos tieto replikoidaan näkymätasolle ja lukitaan tietokannassa, niin tällöin kukaan muu ei voi tehdä muutoksia kyseiseen dataan. Tämä saattaa aiheuttaa ongelmia, jos käyttäjä unohtaa vapauttaa datan tai korkeampi taho haluaa ohittaa muut käyttäjät ja päättää tuotantosuunnitelman. Tiedon lukitseminen on tasapainopiste T6, sillä sen avulla voidaan estää tietokannan pirstaloituminen rinnakkaisessa käytössä, mutta toisaalta se aiheuttaa osaltaan riskin R1.

6.5 Arvioinnin yhteenveto

Skenaarioiden laatiminen toteutettiin yhteistyössä SW-Developmentin tuotanto-osaston työntekijän kanssa, mutta varsinaisen arvioinnin toteutti arkkitehti itse. Tämän takia arvioinnin tuloksiin on syytä suhtautua varauksella, sillä ATAM-prosessin keskeinen idea on, että useat eri henkilöt osallistuvat arviointiin tuoden erilaisia näkökulmia. Jos arkkitehti suorittaa arvioinnin itse, niin arviointi painottuu usein liikaa arkkitehdin suunnittelemiin ominaisuuksiin. Toisaalta tässä työssä arvioinnin onkin tarkoitus olla hyvin suppea, sillä toteutetun arvioinnin keskeinen tavoite on, että arkkitehti voi itse varmistua suunnitelmastaan ennen kuin suunnitelma esitetään laajemmalle yleisölle. [kohta 6.3]

Arvioitujen skenaarioiden osalta voidaan todeta, että tärkeimmät laatuvaatimukset muunneltavuuden ja ylläpidettävyyden suhteen täyttyvät. Keskeisimmät suunnitteluratkaisut kyseisten laatuvaatimusten osalta ovat MVP-mallin käyttö, kerrosten eristäminen ja vuorovaikutus sekä komponenttijako. Datan replikointi ja lukitseminen näkymätasolle parantaa suorituskykyä, mutta toisaalta kyseinen ratkaisu aiheuttaa myös riskin rinnakkaisessa käytössä. Lisäksi datan replikointi luo tasapainopisteen skenaariossa, jossa ohjelma kaatuu odottamattomasta syystä. Tämän perusteella datan replikointiin tulee pohtia vaihtoehtoisia ratkaisuja. MVP-malli todettiin skenaarioiden 2, 3 ja 5 kohdalla hyväksi ratkaisuksi, mutta kyseinen suunnitteluratkaisu liittyy osittain datan replikointiin, jolloin on syytä pohtia muita ratkaisuvaihtoehtoja myös MVP-mallin osalta.

7 YHTEENVETO

Uudelleenkäytöllä on niin taloudellisia, teknisiä kuin organisaatiollisia hyötyjä, mutta komponentit tai tuoterungot eivät synny itsestään. Niiden rakentaminen edellyttää teknistä asiantuntemusta, mutta ennen kaikkea myös koko organisaation tukea ja tahtoa sekä tarkkaa ja täsmällistä määrittelyä ja suunnittelua.

Komponentin toteuttamisessa tärkeimpiä asioita ovat komponenttien korkea itsenäisyyden aste sekä selkeät rajapinnat ja vastuualueet. Jotta komponentteja voidaan hyödyntää tehokkaasti, tulee niiden käyttöä miettiä arkkitehtuuritasolla. Tällöin voidaan avuksi käyttää muun muassa suunnittelumalleja. Tuoterunkojen osalta tulee ymmärtää niiden eri kypsyysasteet ja tuoterunkojen pitkä kehitysprosessi – toimivan tuoterungon rakentaminen vaatii paljon resursseja. Lisäksi tuoterunkojen osalta määrittelyvaihe korostuu entisestään yksittäiseen järjestelmään verrattuna, sillä tuoterungon on tarkoitus palvella useampaa tuoteperheen tuotetta. Tällöin tulee kiinnittää erityshuomiota muunneltavuuteen sekä tuotteiden ominaisuuksiin. Kuten mainittu, tuoterunkojen kehitys on pitkäaikainen prosessi ja tällöin tuoterungon vaikutukset organisaatioon korostuvat, jolloin myös organisaationäkökulma tulee huomioida.

Tuoterungon tai komponenttien rakentaminen voidaan aloittaa erilaisista tilanteista. Eräs tavallinen tilanne on, että organisaatio on rakentanut yhden tai useamman ohjelmiston ja myöhemmin havaitaan, että niissä on mahdollisuutta uudelleenkäyttöön. Tällöin luonnollinen lähestymistapa kohti komponentteja ja tuoterunkoa on määritellä mahdollisesti uudelleenkäytettävät osat. Jos ohjelmistoista ei ole riittävästi ymmärrystä määrittelyn toteuttamiseksi, on ohjelmistoja syytä analysoida vielä tarkemmin, jotta saadaan kokonaisvaltainen ymmärrys ohjelmistoista. Sopivia työkaluja tähän ovat muun muassa UML-kaaviot.

Vaatimusmäärittelyssä on luonnollista pohtia ensin suunniteltavan arkkitehtuurin yleiset vaatimukset, sillä ne osaltaan vaikuttavat arkkitehtuuriratkaisuihin. Käsitemalli on hyvä tapa luoda kehittäjille yhteinen näkemys toimintaympäristöstä. Toiminnallisten vaatimusten määrittelyssä voidaan käyttää apuna vanhempien järjestelmien toiminnallisuutta, mutta luonnollisesti tuoterunko asettaa myös uusia vaatimuksia. Tuoterunkojen kohdalla yksi tärkeimpiä asioita onkin muunneltavuusvaatimusten määrittely, johon voidaan käyttää apuna esimerkiksi muunneltavuusmatriisia tai piirremallia.

Tämän työn tuloksena suunniteltiin tuoterunkoajattelua hyödyntävä komponenttiarkkitehtuuri. Suunnittelun lähtökohtana oli organisaation ja aikaisempien järjestelmien asettamat vaatimukset. Eräs vaatimus oli, että suunniteltavassa järjestelmässä hyödynnetään .NET-kehitysympäristöä, jolloin luonnollinen lähtökohta on hyödyntää .NET-kerrosarkkitehtuurityyliä. Kyseistä kerrosarkkitehtuurityyliä ei kuitenkaan otettu

sellaisenaan käyttöön, vaan arkkitehtuurityyliä muokattiin organisaatiolle sopivaksi. Arkkitehtuurisuunnitelman keskeisimpiä piirteitä ovat kerrosten tiukasti määritelty vuorovaikutus (strict top-down interaction), kerrosten väliset rajapinnat, toimintalogiikan jakaminen komponentteihin, MVP-mallin hyödyntäminen näkymätasolla, suunnittelumallien käyttö (kuten fasaadi, adapteri sekä mediaattori) ja varioituvuuden sekä uudelleenkäytön huomioiminen kaikilla kerroksilla.

Suunniteltu arkkitehtuuri arvioitiin ATAM-prosessia hyödyntäen. Täysimittaista ATAM-arviointia ei kuitenkaan toteutettu vaan suppeampi versio, sillä suppeampi versio sopii työn laajuuteen paremmin. ATAM-prosessin perusteella suunniteltu arkkitehtuuri pystyy toteuttamaan skenaariot melko hyvin. Prosessin perusteella havaittiin kuitenkin, että muun muassa datan tallennukseen ja käsittelyyn tulee kiinnittää enemmän huomiota. Lisäksi on hyvä pohtia tarkemmin mahdollisia tulevaisuuden toiminnallisia vaatimuksia, eikä peilata toiminnallisuutta liikaa vanhemmista järjestelmistä.

Työn tuloksena suunniteltu arkkitehtuuri antaa osaltaan lähtökohdan laajemmalle tuoterunkoa ja komponenttitekniikoita hyödyntävälle ohjelmistolle. Arkkitehtuuria ja ohjelmistoa tulee kehittää jatkossa vaiheittain eteenpäin, jotta tulevaisuudessa uudelleenkäytön potentiaali saadaan hyödynnettyä. Lisäksi tämä työ sisältää hyödynnettävää tietoa komponenteista, tuoterungoista ja niihin liittyvistä asioista sekä myös analysointia vanhemmista järjestelmistä, joiden dokumentaatio on puutteellinen.

Laajemmin ajateltuna tuoterungon ja komponenttien hyödyntämisestä on kyse investoinnista ja investoinnit sisältävät riskejä. Toisin kuin esimerkiksi laiteinvestoinnissa, on tuoterungoissa ja komponenteissa hankinta- ja kehitysaika tavallisesti pidempi eikä esimerkiksi tuoterungon suunnittelua voi kovin helposti myydä pois, jos se havaitaan toimimattomaksi. Lisäksi takaisinmaksu ei suinkaan ole itsestäänselvyys. Tämän takia on erittäin tärkeää, että mikäli investointi tuoterunkoon ja komponentteihin tehdään, se tehdään kunnolla.

LÄHTEET

[1] Haverila, M.J., Uusi-Rauva, R., Kouri, I., Miettinen, A. Teollisuustalous. 5. painos. Tampere 2005, Infacs Oy. 510 p.

[2] Bertolino, A., Mirandola, R. Towards Component-Based Software Performance Engineering. 6th ICSE Workshop on Component-Based Software Engineering: Automated Reasoning and Prediction. Portland, Oregon, USA May 3-4, 2003, Carnegie Mellon University, USA, Monash University, Australia.

[3] Sommerville, I. Software Engineering, Eight Edition. 2006, Addison-Wesley. 864 p.

[4] Koskimies, K., Mikkonen, T. Ohjelmistoarkkitehtuurit. Helsinki 2005, Talentum Media Oy. 250 p.

[5] McIlroy, D. 'Mass Produced' Software Components. NATO Software Engineering Conference 1968. Garmisch, Germany, October 7-11, 1969. pp. 138-156.

[6] Szyperski, C., Gruntz, D., Murer, S. Component Software Beyond Object-Oriented Programming. 2. painos. Iso-Britannia 2002, Pearson Education Limited. 624 p.

[7] Crnkovic, I., Larsson, M. Building Reliable Component-Based Software Systems. 1. painos. Massachusetts 2002, Artech House. 454 p.

[8] Atkinson, C., Bayer, J., Bunse, C., Kamsties, E., Laitenberger, O., Laqua, R., Muthig, D., Paech, B., Wüst, J., Zettel, J. Product Line Engineering with UML. Iso-Britannia 2002, Pearson Education Limited.

[9] Brooks, F.P. No Silver Bullet: Essence and Accident in Software Engineering. Computer, 20(1987)4. pp. 10-19.

[10] Veryard, R. Designing Software Components. 1999, SCIPIO Consortium. [viitattu 8.9.2011] Saatavissa: http://www.users.globalnet.co.uk/~rxv/scipio/swpdsc.pdf

[11] Rintala, M., Jokinen, J. Olioiden ohjelmointi C++:lla. 4. painos. Helsinki 2005, Talentum Media Oy. 377 p.

[12] Weyuker, E.J. Testing Component-Based Software: A Cautionary Tale. IEEE Software 15(1998)5.

[13] Microsoft Patterns & Practices Team. 2009. Microsoft Application Architecture Guide. 2. painos. Yhdysvallat, Microsoft Press. 560 p.

[14] Laukkonen, A. Tehokkuudensuunnittelutyökalun tuoterunkoarkkitehtuuri. Diplomityö. Tampere 2007. Tampereen teknillinen yliopisto, automaatiotekniikan koulutusohjelma. 64 p.

[15] Löwy, J. Programming .NET Components. 2. painos. Gravenstein Highway North, Sebastopol, CA, USA, 2005, O'Reilly Media. 648 p.

[16] Gamma, E., Helm, R., Johnson, R., Vlissides, J. Design Patterns: Elements of Reusable Object-Oriented Software. Portland, OR, USA, 1995, Addison-Wesley. 416 p.

[17] Järvinen, J. Komponenttiohjelmointi. Hermia, Tampere 2011. Moonsoft Oy. Konsulttiesitelmä. 18 p.

[18] Jifeng, H., Li, X., Liu, Z. Component-Based Software Engineering – the Need to Link Methods and their Theories. Macao 2005, The United Nations University – International Institute for Software Technology, UNU-IIST Report No. 330. 32 p.

[19] McConnell, S.C. Code Complete. 2. painos. Redmond, Washington, 2004 Microsoft Press. 914 p.

[20] McKinley, P.K., Sadjadi, S.M., Kasten, E.P., Cheng, B.H.C. Composing Adaptive Software. Computer 4(2005)1. pp. 56-64.

[21] Vuorinen, M. LINQ-ohjelmointikieleen yhdistetty hakuarkkitehtuuri. Insinöörityö. Helsinki, 2010. Metropolia Ammattikorkeakoulu, tietotekniikan koulutusohjelma. 100 p.

[22] Vanderlei, T.A. Risks and Challenges of component-based development. 2004, Centro de Informática, Universidade Federal de Pernambuco. [viitattu 8.9.2011] Saatavissa: www.cin.ufpe.br/~in1045/slides/app12.ppt

[23] Clements, P., Northtop, L. Software Product Lines. 2003, Carnegie Mellon University Software Engineering Institute. [viitattu 8.9.2011] Saatavissa: http://www.cs.helsinki.fi/u/vjkuusel/gradu/vanhat/Software%20Product%20Lines.pdf

[24] Bosch, J. Maturity and Evolution in Software Product Lines: Approaches, Artefacts and Organization. Groningen, Alankomaat, 2002, University of Groningen Department of Computing Science. [viitattu 8.9.2011] Saatavissa: http://www.janbosch.com /Articles/SPLC2BoschFinal.pdf

[25] Pohl, K., Metzger, A. Variability Management in Software Product Line Engineering. Shanghai 2006, University of Limerick, University of Duisburg-Essen. [viitattu 8.9.2011] Saatavissa: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.101.6885&rep=rep1&type=pdf

[26] Pohl, K., Böckle, G., van der Linder, F. 1. painos. Software Product Line Engineering: foundations, principles and techniques. Saksa 2005, Springer. 468 p.

[27] Conway, M.E. How Do Committees Invent?. Datamation 14(1968)4. pp. 28-31.

[28] Hyvönen, E. Ohjelmistoliiketoiminta. 1. painos. Vantaa 2003, Dark Oy. 248 p.

[29] Bosch, J. Software Product Lines: Organizational Alternatives. Groningen, Alankomaat, 2001, University of Groningen Department of Computing Science. [viitattu 8.9.2011] Saatavissa: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.27.6302&rep=rep1&type=pdf

[30] Schmid, K., Verlage, M. The Economic Impact of Product Line Adaptation and Evolution. IEEE Software July/August 2002. pp. 50-57.

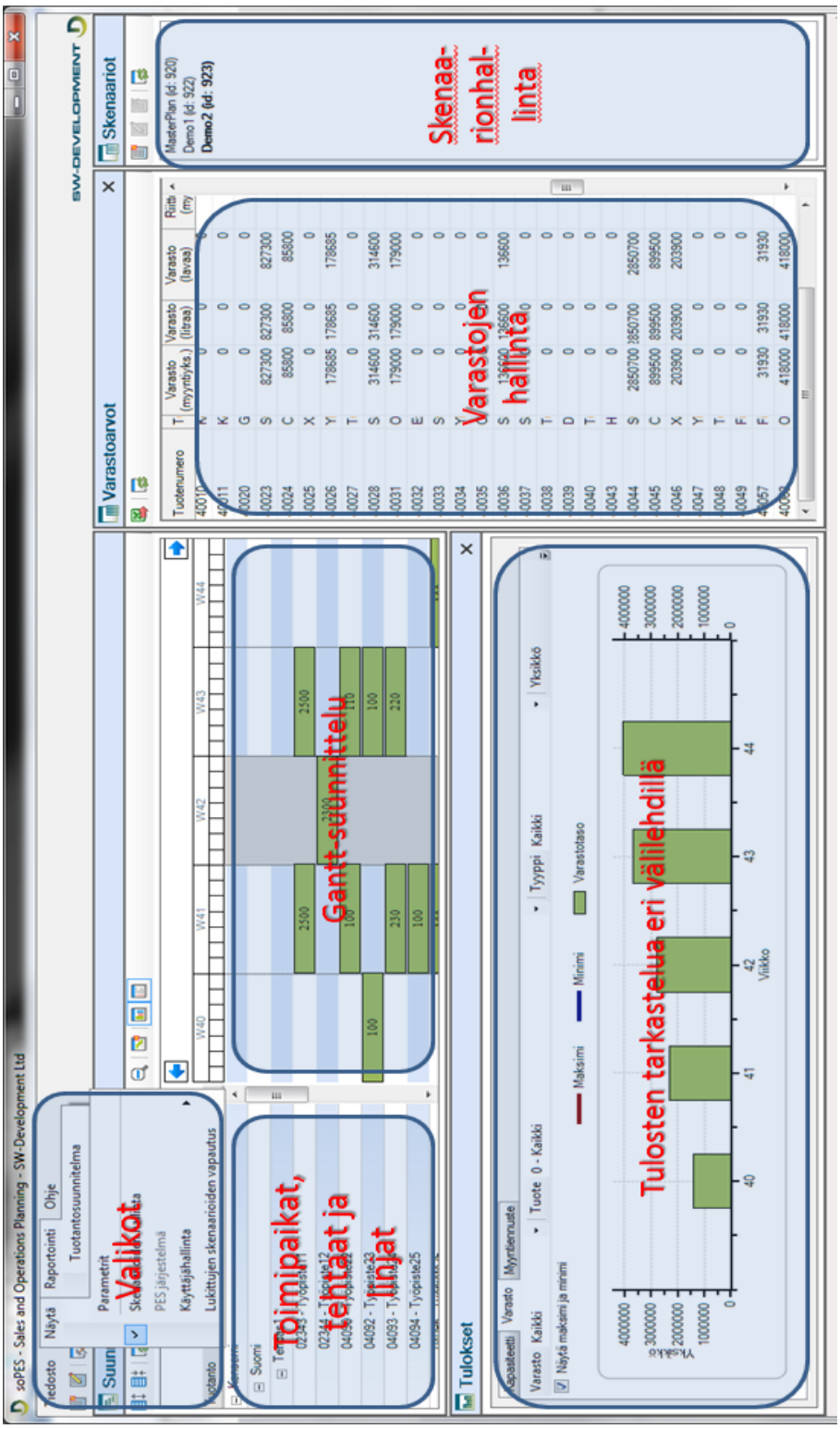
[31] Eriksson, M. An Introduction to Software Product Line Development. Örnsköldsvik, Ruotsi, 2003, Alvis Hägglunds AB. [viitattu 12.9.2011] Saatavissa: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.63.9209&rep=rep1&type=pdf

[32] Halonen, A. 2011. Diplomi-insinööri, Technology Specialist. SW-Development. Tampere. Haastattelu 7.10.2011.

[33] Bass, L., Clements, P., Kazman, R. Software Architecture in Practice. 10. painos. Westford, MA, USA 2007, Carnegie Mellon Software Engineering Institute. 528 p.

[34] Kunnari, M. 2011. Diplomi-insinööri, Business Intelligence Speacialist. SW-Development. Tampere. Palaveri 14.11.2011.

LIITE 1: KÄYTTÖLIITTYMÄKUVA



LIITE 2: ATAM-LAATUPUU

Suluissa olevat arvot ennen skenaariota kuvaavat skenaarion tärkeyttä ja vaikeutta asteikolla 1-3 (1 = vähiten tärkeä tai vaikea ja 3 = eniten tärkeä tai vaikea). Esimerkiksi (1,3) tarkoittaa, että skenaario ei ole tärkeä, mutta arkkitehtuurin kannalta vaikea.

- **SUORITUSKYKY**
  - Simulointiajo
    - (2,1) Asiakas haluaa simuloida tuotantosuunnitelman, vasteaika < 30s.
  - Vasteaika
    - (1,1) Käyttäjä haluaa luoda raportin tuotantosuunnitelmasta, vasteaika < 1s.
    - (3,1) Käyttäjä muokkaa skenaariota ja haluaa tallentaa skenaarion tietokantaan, vasteaika < 3s.
  - Optimointi
    - (1,1) Käyttäjä haluaa aikatauluttaa tuotantosuunnitelman uudelleen optimoinnin avulla jollekin linjalle lennossa, vasteaika < 1s.
    - (1,1) Käyttäjä haluaa optimoida tuotantosuunnitelman saaden useita eri vaihtoehtoja, vasteaika < 1s / optimointiajo.
- **LUOTETTAVUUS**
  - Tulosten luotettavuus
    - (3,1) Käyttäjä laatii tuotantosuunnitelman ja hyväksyy sen toteutettavaksi.
    - (3,1) Käyttäjä optimoi tuotantosuunnitelmaa valmiilla optimointikomponentilla ja haluaa optimoidun tuloksen käyttöön.
  - Tuotantosuunnitelman teko
    - (3,3) Ohjelman kaatuu odottamattomasta syystä, mutta käyttäjän muokkaamaa tai luomaa data ei kuitenkaan katoa.
- **SAATAVUUS**
  - Rinnakkaisuus
    - (3,3) Useat ihmiset haluavat käsitellä samaa dataa rinnakkain.
  - Verkkoyhteys katkeaa
    - (1,2) Asiakas haluaa käyttää järjestelmää ilman yhteyttä tietokantaan.
- **TURVALLISUUS**
  - Tietoturva
    - (1,2) Käyttäjän tietokone hajoaa eikä se saa aiheuttaa ongelmia järjestelmään.
    - (1,3) Asiakkaan kilpailija vakoilee tietoliikennettä.
    - (3,1) Esimies yrittää katsoa toisten esimiesten tietoja ilman lupaa.
  - Käyttäjäturvallisuus
    - (3,2) Kesätyöntekijä käyttää järjestelmää ja muuttaa vahingossa tuotantosuunnitelmaa.
- **YLLÄPIDETTÄVYYS**
  - Uusi näkymä
    - (1,1) Asiakas haluaa järjestelmään lisäominaisuutena graafisia näkymiä toteutuneista tilauksista.
  - Uusi komponentti
    - (1,1) Järjestelmään toteutetaan uusi optimointikomponentti.

    - Käyttöympäristön muutokset
        - (1,2) Asennusympäristö muuttuu palvelinajosta paikalliseen ajoon.
- **MUUNNELTAVUUS**
    - Ulkoisen systeemin muutos
        - (1,2) Asiakas vaihtaa tietokannan Oraclen tietokannasta Microsoftin tietokantaan.
    - Uudet ominaisuudet
        - (1,2) Asiakkaan järjestelmässä ei ole simulointimahdollisuutta, mutta asiakas haluaa ostaa ominaisuuden järjestelmän toimituksen jälkeen.
- **SIIRRETTÄVYYS**
    - Asiakas vaihtaa tietokonetta
        - (2,1) Asiakas ostaa uuden tietokoneen ja haluaa asentaa järjestelmän siihen.
- **KÄYTETTÄVYYS**
    - Lokalisointi
        - (2,1) Käyttäjä haluaa vaihtaa järjestelmän kielen suomesta englanniksi.
    - Miellyttävyys
        - (1,1) Asiakas haluaa itse päättää visuaalisen ilmeen.
    - Tiedonhaku
        - (2,1) Asiakas haluaa hakea dataa tietokannasta sekä XML-tiedostosta.
    - Yhdenmukaisuus
        - (3,1) Käyttäjä voi luottaa järjestelmän peruslogiikan olevan sama eri osissa ohjelmistoa.
- **VARIOITAVUUS**
    - Eri tuoteominaisuudet
        - (1,2) Järjestelmä on asiakkaan käytössä, mutta asiakas haluaisi samankaltaisen järjestelmän tytäryhtiölle ilman simulointi- ja optimointimahdollisuutta kustannusten säästämiseksi.
    - Eri käyttäjäryhmät
        - (3,2) Asiakas haluaa itse luoda uudet käyttäjäoikeudet työnjohtajille ja toimitusjohtajalle.